Please type a plus sign (+) inside this box → [+] 5-26-00

PTO/SB/16 (2-98)
Approved for use through 01/31/2001. OMB 0651-0037
Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53 (c).

05/25/00 JC541 U.S. PTO 60/207891 05/25/00

INVENTOR(S)

Given Name (first and middle [if any])	Family Name or Surname	Residence (City and either State or Foreign Country)
Michael W.	Masters	Fredericksburg, VA
Paul V.	Werme	Dahlgren, VA
Lonnie R.	Welch	Athens, OH

[ ] *Additional inventors are being named on the ___ separately numbered sheets attached hereto*

TITLE OF THE INVENTION (280 characters max)

A Method And Apparatus For Resource Management

CORRESPONDENCE ADDRESS

Direct all correspondence to:

[X] Customer Number 23501 → *Place Customer Number Bar Code Label Here* 23501

OR *Type Customer Number here*

PATENT TRADEMARK OFFICE

[X] Firm *or* Individual Name	James B. Bechtel, Esq.				
Address	Naval Surface Warfare Center (Code CD222)				
Address	17320 Dahlgren Road				
City	Dahlgren	State	VA	ZIP	22448-5100
Country	United States	Telephone	540-653-8061	Fax	540-653-7816

ENCLOSED APPLICATION PARTS *(check all that apply)*

[X] Specification *Number of Pages* 279 [ ] Small Entity Statement

[ ] Drawing(s) *Number of Sheets* [ ] Other (specify)

METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT *(check one)*

[ ] A check or money order is enclosed to cover the filing fees

[X] The Commissioner is hereby authorized to charge filing fees or credit any overpayment to Deposit Account Number: 50-0967

FILING FEE AMOUNT ($): $150.00

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

[ ] No.

[X] Yes, the name of the U.S. Government agency and the Government contract number are: DEPT. OF THE NAVY - NAVAL SURFACE WARFARE CENTER

Respectfully submitted,

SIGNATURE James B Bechtel Date 5/25/00

TYPED or PRINTED NAME James B. Bechtel REGISTRATION NO. *(if appropriate)* 29,890

TELEPHONE 540-653-8061 Docket Number: NC82185

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

This collection of information is required by 37 CFR 1.51. The information is used by the public to file (and by the PTO to process) a provisional application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 8 hours to complete, including gathering, preparing, and submitting the complete provisional application to the PTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, Washington, D.C., 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Box Provisional Application, Assistant Commissioner for Patents, Washington, D.C., 20231.

A/DROV

Description of Resource Management, NSWCDD patent case number TBD
Michael W. Masters, NSWCDD, Code B35

Resource Management consists of a set of cooperating computer programs that provides an ability to dynamically allocate computing tasks to a collection of networked computing resources (computer processors interconnected on a network) based on the following measures: an application developer/user description of application computer program performance requirements; measured performance of each application programs; measured workload (CPU processing load, memory accesses, disk accesses) of each computer in the network; and measured inter-computer message communication traffic on the network.

The capabilities provided by Resource Management are as follows:

- Dynamically allocate computer programs to computers within a network based on a user statement of computer program performance goals
- Dynamically change allocation according to changing system loading conditions
- Change allocations based on manual operator direction
- Dynamically adjust to overall computer workload by balancing processing loads among a number of scalable, replicated load sharing programs
- Dynamically compensate for computer failures and network link failures by restarting copies of lost computer programs on surviving computers within the network

Resource management consists of the following computer program components:

- A Performance Specification Language whereby application developers/users define the performance goals they want Resource Management to insure for each application. Application computer program performance requirements, or performance goals, consist of requested CPU execution times for each application. A performance goal may also be specified for the end-to-end processing time of a combination of several computer programs which are designed to process data in a sequence (referred to as a path). In a path, each computer program in sequence performs a defined set of processing steps and then passes its data to the next computer program in the path.

- A Specification Language Processor Program that converts application developers/users requirements into instruction for action by the remainder of Resource Management

- An Operating System Instrumentation Subsystem that collects measured performance data from each computer in the network. This subsystem consists of two types of components. The first is an Operating System Instrumentation Data Collector Program, a copy of which runs on each program in the network and collects computer performance data from the operating system on which it resides. The second is a centralized Operating System Measurement Repository Program that accumulates operating system instrumentation data from all the collector programs. The collector programs periodically report the data they have collected to the central operating system measurement repository program.

- A System Health Monitor Subsystem, consisting of a heartbeat mechanism (periodic messages to all computers in the network). The System Health Monitor Subsystem detects the failure of any computer in the network or the loss of a network link within the overall network and reports this information to the Operating System Instrumentation Subsystem.

- An Application Instrumentation Subsystem that collects measured performance data from each application running under the scope and control of Resource Management. This subsystem consists of two types of components. The first is an Application Program Instrumentation Data Collector Program, a copy of which runs on each program in the network and collects computer performance data from the application computer programs running on the computer on which the collector program resides. The second is a central Application Program Measurement Repository Program that accumulates application instrumentation data from all the collector programs. The

collector programs periodically report the data they have collected to the central application measurement repository program.

- A Resource Allocation Program that utilizes measurement information from both the Operating System Measurement Repository Program and the Application Program Instrumentation Data Collector Program to make decisions concerning the allocation or assignment of computer programs to computers within the network. It compares the observed performance of each application program with the application developer/user requested performance level. For each application, if the application's performance is within bounds specified by the application developer/user, the resource allocation program makes no change of allocation to the system (computers, network and applications). If one or more applications are found to be performing in a less than satisfactory manner compared to the performance goals specified by the application developer/user, or if based on trend analysis they are projected to begin performing in a less than satisfactory manner, or if a computer failure or network link failure has been detected in the network, then the Resource Allocation Program examines data on the measured loading and performance of each computer in the network from the operating system instrumentation data collector program, applies an optimization algorithm, and selects a configuration change, or application computer program reassignment re-assignment to a different computer designed to restore the application's performance to the level specified by the application developer/user. The Resource Allocation Program sends the configuration change request to a Program Control Subsystem and its agents for implementation, (see description of program control component below). The Resource Allocation Program selects one of the following actions:

    - If the computer program that is not meeting performance goals has been designed as a scalable, replicated load-sharing computer program, then the Resource Allocation Program will select a computer from the network which has sufficient reserve capacity to provide adequate processing services and will direct the Program Control Subsystem to load and initialize a second (and eventually a third, and a fourth, etc.) copy of the application program that is not meeting its performance goals.

    - If the program that is not meeting its performance goals is not a scalable, replicated load-sharing program, then the Resource Allocation Program will direct that the Program Control Subsystem move it to a different computer. This move operation consists of starting a new copy of the application program that is not meeting its performance goal on a computer with the reserve capacity to run the program satisfactorily and then shutting down the copy of the application program that is not meeting its performance goals.

    - If a computer or network link has failed, then the Resource Allocation Program selects one or more computers in the network with the capacity to run the applications on the computer or computers that have failed or that have been isolated from the rest of the network by the failure of the network link. It will direct the Program Control Subsystem to load and initialize copies of all application programs that have been rendered inoperable by the computer failures or network link failure.

- A Program Control Subsystem that receives resource allocation configuration changes from the Resource Allocation Program and carries them out. The Program Control Subsystem consists of a Program Control Program and a set of Program Control Agents, one of which resides on each computer in the network. The Program Control Program has two modes of operation: a manual, Program Control Program Operator activated mode and an automatic mode commanded by the Resource Allocation Program. When the Program Control Program receives a configuration change directive, either from the Program Control Program Operator or the Resource Allocation Program, it sends a command to the Program Control Agent on the computer where the configuration change operation is to take place. The Program Control Agent on that computer performs the appropriate action by means of interaction with the operating system of the computer on which it resides and by means of interaction with the file system of the computer network.

- If the requested configuration change results in starting a new program on the designated computer, then the Program Control Agent sends commands to the file system causing the new program to be loaded across the network and initiated on the designated computer.

- If the requested configuration change results in shutting down a program on the designated computer, then the Program Control Agent sends commands to the operating system causing the program to be stopped.

Based on long-term oversight and technical direction of the Resource Management capability from its inception as a part of the joint DARPA and Navy funded HiPer-D program and the DARPA follow-on Quorum program, it is my assessment that three individuals have contributed substantially to invention of the concept and architecture of Resource Management. The initial concept and design were developed by the author, Michael W. Masters, and by Dr. Lonnie Welch while he was on sabbatical at NSWCDD as a visiting professor. Subsequently, Mr. Paul Werme added substantial technical detail to the architecture. Two individuals have been predominant in the detailed design of the implementation of the components of Resource Management described above and in the demonstration and verification that the Resource Management concept is realizable. These are Dr. Lonnie Welch and Mr. Paul Werme. In addition, Mr. Larry Fontenot may have contributed substantially to the invention of the Performance Specification Language and the Specification Language Processor Program.

This assessment, along with the technical accuracy and completeness of the description provided above, is solely that of the author and should be considered preliminary subject to review and clarification by Dr. Welch and Mr. Werme. To the best of the author's knowledge, all work on Resource Management, from its inception, has been performed either by government employees or by non-government employees working under the direction of government employees through government contracts.

EXECUTIVE SUMMARY 1
1.0 INTRODUCTION 3
1.1 HiPer-D Phase 1 – DARPA Technology Evaluation 3
1.1.1 Phase I Integrated Demonstration One (I1) 3
1.2 HiPer-D Phase 2 – DARPA/COTS Technology and Critical Issues Evaluation 4
1.2.1 HiPer-D Phase 2 Engineering Testbed One (T1) Demonstration 6
1.2.2 HiPer-D Phase 2 Engineering Testbed Two (T2) Demonstration 7
1.2.3 HiPer-D Phase 2 Engineering Testbed Two A (T2A) Demonstration 7
1.2.4 HiPer-D Phase 2 Engineering Testbed Three (T3) Demonstration 8
1.3 Demo 98 Objectives 9
2.0 STAND-ALONE ENGINEERING TESTS 11
2.1 Evaluating the Performance of Multicast Communications 11
2.2 Data Distribution Experiment 11
2.3 Windows NT Investigations 12
3.0 ADVANCED COMPUTING TESTBED DEMO 98 INTEGRATED DEMONSTRATION DESCRIPTION 13
3.1 AAW Subsystem Functional Description 17
3.1.1 Advanced Track Correlation and Filtering (ATCF) 19
3.1.1.1 ATCF Overview 19
3.1.1.1.1 Standard Message Format 20
3.1.1.1.2 MFAR Broker 20
3.1.1.1.3 IP Multicast Communications 21
3.1.1.1.4 ATCF Fault Tolerance 24
3.1.1.1.5 Track Number Mapping 25
3.1.2 Air Engagement Control (AEC) 25
3.1.2.1 AEC Component Summary 26
3.1.2.2 AEC Display 27
3.1.2.3 Display State Data Server 28
3.1.2.4 Manual Engage Control 28
3.1.2.5 Plan Server 28
3.1.2.6 Semi-Auto 29
3.1.2.7 Auto-SM 29
3.1.2.8 Auto-Special 29
3.1.2.9 Engagement Server 30
3.1.3 Track Data Services Components 34
3.1.3.1 Radar Track Data Server (RTDS) 35
3.1.3.2 CORBA Track Number Server (CTNS) 35
3.1.3.2.1 CTNS Overview 36
3.1.3.2.2 Overall Architecture 37
3.1.3.2.3 CTNS / TNSS Client Communications 38
3.1.3.2.4 CTNS Group Communication 39
3.1.3.2.5 Startup Processing 40
3.1.3.2.6 Performance 41
3.1.3.3 Sensor Rate Server (SRS) 41
3.2 Land Attack and C^4I Subsystem Functional Description 41
3.2.1 Advanced Tomahawk Weapons Control System (ATWCS) 42

3.2.2 Joint Maritime Command Information System (JMCIS) 47
3.2.2.1 DIS to OTHGOLD Converter 48
3.2.3 Advanced Battle Management and Execution (ABMX) System 49
3.2.4 Data Brokers – Legacy System Interface 49
3.2.4.1 JMCIS/AACT/AAW Interface 50
3.2.4.2 Real-Time Data AAW Track Path 50
3.2.4.3 OTH Track Data Path 51
3.2.4.4 Aegis Air Correlator Tracker (AACT) 51
3.3 Simulation and Support Components 52
3.3.1 Environmental Simulations (EnvSims) 54
3.3.1.1 Entity Simulations 55
3.3.1.2 Sensors Simulations 58
3.3.1.3 Displays 59
3.3.2 Simulation Control (SIMCON) 63
3.3.2.1 Modifications Description 63
3.3.2.2 Restrictions 63
3.3.3 Kinematics Daemon (KINED) 64
3.3.4 Weapons Control System Simulator (WCS Sim) 64
3.3.5 Identification Upgrade Simulator (IDU Sim) 64
3.3.6 NSFS Simulator (NSFSsim) 64
3.3.7 Digital Call For Fire Support Components 65
3.3.7.1 Remote Digital Data Link (RDDL) 65
3.3.7.2 TACFIRE Processor 65
3.3.7.3 C3I Broker 65
3.3.8 System Control 66
3.3.9 Clock Synchronization 66
3.3.10 Near Real-time Data Collection/Display (JEWEL) 67
3.3.11 Group Communications 69
3.4 Resource Management. 69
3.4.1 System Monitoring. 73
3.4.1.1 UNIX Operating System and Network Monitoring. 73
3.4.1.1.1 Methodology. 74
3.4.1.2 Windows NT Operating System and Network Monitoring. 76
3.4.1.2.1 Windows NT Statistics Retrieval 76
3.4.1.2.2 Network Interface. 77
3.4.1.3 Monitoring Status and History Servers. 77
3.4.2 Dynamic Resource Management. 78
3.4.2.1 System Model. 79
3.4.2.2 Adaptive QoS and Resource Management. 80
3.4.2.2.1 Path QoS Monitor. 81
3.4.2.2.2 QoS Diagnosis. 82
3.4.2.2.3 Resource QoS Monitor. 82
3.4.2.2.4 Resource Allocation. 83
3.4.2.3 Results. 83
3.4.3 Resource Control / Program Control. 83
3.4.3.1 Graphical User Interface. 83

3.4.3.2 Subsystem Managers. 88
3.4.3.3 Host Agents. 88
3.4.3.4 Summary. 89
3.4.4 QoS and System Specifications. 90
3.4.5 Visualization. 90
3.4.5.1 Host Display. 90
3.4.5.1.1 Host Display Design. 92
3.4.5.1.2 Data Formats. 92
3.4.5.1.2.1 Host Configuration File. 92
3.4.5.1.2.2 Interface to Data Server. 93
3.4.5.1.2.2.1 Host Configuration Message. 93
3.4.5.1.2.2.2 Host Process Message. 94
3.4.5.1.3 Graph Display Interface. 95
3.4.5.1.4 User Interface. 95
3.4.5.2 Path Display 96
3.4.5.2.1 Path Display Design. 96
3.4.5.2.2 Data Display 96
3.4.5.2.2.1 Data Flow 97
3.4.5.2.2.2 Application and Path Performance Data 98
3.4.5.2.3 User Interface. 99
3.4.5.3 Resource Management Decision Review Display 100
3.4.5.3.1 Design. 100
3.4.5.3.2 Data Formats. 101
3.4.5.3.2.1 Event Message. 101
3.4.5.3.2.2 Scaleup Message. 103
3.4.5.3.3 User Interface. 104
3.5 Demo 98 Hardware Configuration 104
3.6 Demo 98 Scenario 106
3.7 Integrated System Demonstration 107
3.7.1 Environmental Simulation 107
3.7.2 ATWCS Launch Control Real Time Group 111
3.7.3 Fault Tolerant Engagement Server 113
3.7.3.1 Fault Injection Control 115
3.7.3.2 Fault Recovery and Performance Impact 116
3.7.3.3 Summary and Future 120
3.7.4 Digital Call for Fire (CFF) 120
3.7.4.1 FO/FAC Subsystem 121
3.7.4.2 CFF Initiation Sequence 125
3.7.4.3 Visual Deconfliction 125
3.7.4.4 OTH Track Injection 125
3.7.4.5 CFF Engagement Transmission 126
3.7.4.6 Engagement Sequence 126
3.7.5 Demo 98 Resource Management Scenario 130
3.7.5.1 Overview. 130
3.7.5.2 Fault Tolerance of Resource Management Components 131
3.7.5.3 Control of Application Scalability. 134

3.7.5.4 Application Fault Detection and Recovery. 138
3.7.5.5 Summary. 139
4.0 LESSONS LEARNED 140
4.1 CORBA Plan Server Lessons Learned 140
4.1.1 Advantages Offered by the CORBA Technology 140
4.1.2 CORBA Learning Curve 140
4.1.3 Difficulties with Legacy Systems: 141
4.1.4 CORBA Specifications versus Available ORB Implementations 141
4.1.5 CORBA in Perspective 141
4.2 CORBA TNS Lessons Learned 142
4.3 Engagement Server Lessons Learned 143
4.3.1 Synchronization and Determinism 144
4.3.2 Cross-Group Data Difficulties 146
4.3.3 Recovery Time and Group Coupling 147
4.3.4 Precise Fault Injection 148
4.4 Remote Digital Call for Fire (CFF) Lessons Learned 148
4.5.1 Monitoring 149
4.5.2 Resource Management Decision Making 150
4.5.3 Resource Control / Program Control 151
4.5.4 System and Software Specifications 151
4.5.5 Visualization 151
4.5.6 Summary 152

Ex.1009 / Page 8 of 280
TESLA, INC.

APPENDIXES

A – Evaluating The Performance of Multicast Communications
B – HiPer-D Data Distribution Experiment
C – Windows NT Investigations
D – History of HiPer-D Track Correlator and Filtering Process
E – Group Communications
F – Adaptive QoS and Resource Management Using A *Posteriori* Workload Characterization
G – Resource Management QoS and System Specifications

EXECUTIVE SUMMARY

The High Performance Distributed (HiPer-D) Computing Project was a six year joint Defense Advanced Research Projects Agency (DARPA)/Aegis Program Office (NAVSEA PMS 400) program to investigate the application of advanced technologies and concepts to the Naval Surface Ship Anti-Air Warfare (AAW) problem domain. The first phase of the program, funded primarily by DARPA, was involved in prototype development and the evaluation and demonstration of selected DARPA technologies using the system-level prototype developed. The second phase, funded primarily by PMS 400, involved prototype expansion and enhancement, incorporating COTS technology as well as DARPA technologies, and a focus of risk reduction of Aegis identified critical technology and system issues. The HiPer-D Computing Project was to have ended with publication of the Engineering Testbed Three (T3) Report. However, due to the success of the project, the Aegis Program Office and DARPA have elected to continue the effort. The next demonstration following T3 was named Demo 98.

The HiPer-D Computing Program Advanced Computing Testbed Demonstration 98 (Demo 98) was held in September 1998 in the System Control Laboratory (SCL) at NSWCDD. Among the attendees were CDR Stevenson of PMS 400 and Dr. Gary Koob of DARPA. Once again, the innovative expertise of the Aegis HiPer-D engineering team at NSWCDD and JHU/APL, coupled with commercial tools and technology, resulted in both the building and experimental validation of 21st century computing architectures that support mission-critical weapon systems. All the following technical objectives were met.

(1) To expand the land attack capability by integrating the Advanced Tomahawk Weapons Control System (ATWCS) Launch Control Real Time (LCRT) group into the HiPer-D testbed.

(2) To enhance the NSFS call for fire from voice communications to a digital capability.

(3) To complete the development and integration of a fully fault tolerant, scalable AAW path through the system. This involved the development of a fault tolerant Engagement Server and a fault tolerant Track Correlator and Filter (TCF).

(4) To enhance the resource management capability of the system. This included increased functionality, reduced fault recovery times, and starting and re-starting of non-HiPer-D components, such as ATWCS.

(5) To further the evolution toward Navy open systems. This included the expansion of the ATM network and the addition of IP multicast capability over all three networks (ATM, FDDI, Ethernet).

(6) To begin assessing the feasibility of using COTS-based distributed object computing technologies for the Navy. This included the integration of CORBA in the Track Number Server (TNS) and the Plan Server components.

(7) To evolve toward long term middleware. This included the integration of commercial products, CORBA-based and NDDS, and the integration of SPREAD group communication in the Advanced Track Correlator and Filter (ATCF).

(8) To expand the simulation capability of the testbed by adding a physics-based, DIS compatible wrap-around simulation capability. The DIS capability would provide the basis for integrating geographically dispersed laboratories into the HiPer-D testbed.

(9) To include navigation/gyro data distribution. The ATWCS LCRT component requires a 16 Hz navigation data input which was delivered over the ATM network.

This report describes the components and conduct of Demo 98, how these objectives were met, and the lessons learned from the effort.

Ex.1009 / Page 11 of 280
TESLA, INC.

1.0 INTRODUCTION

a. The High Performance Distributed (HiPer-D) Computing Project was a six year joint Defense Advanced Research Projects Agency (DARPA)/Aegis Program Office (NAVSEA PMS 400) program to investigate the application of advanced technologies and concepts to the Naval Surface Ship Anti-Air Warfare (AAW) problem domain. The first phase of the program, funded primarily by DARPA, was involved in prototype development and the evaluation and demonstration of selected DARPA technologies using the system-level prototype developed. The second phase, funded primarily by PMS 400, involved prototype expansion and enhancement, incorporating COTS technology as well as DARPA technologies, and a focus of risk reduction of Aegis identified critical technology and system issues. The HiPer-D Computing Project was to have ended with publication of the Engineering Testbed Three (T3) Report. However, due to the success of the project, the Aegis Program Office and DARPA have elected to continue the effort.

1.1 HiPer-D Phase 1 – DARPA Technology Evaluation

a. Phase 1 of the HiPer-D Program began in June of 1991 and ended in 1994. The primary goals were to demonstrate and test DARPA-developed technologies, and assess the viability of including distributed computing in future combat system planning. Major objectives for HiPer-D Phase 1 were to:

(1) Evaluate DARPA technologies for use in combat systems.

(2) Educate Aegis engineers in distributed computing principles and methods.

(3) Provide feedback to Aegis and DARPA to focus further technology development efforts.

1.1.1 Phase 1 Integrated Demonstration One (I1)

a. The DARPA technologies were thoroughly tested through various stand-alone demonstrations and the HiPer-D Phase 1 Integrated Demonstration One (I1) that occurred in March 1994. I1 successfully integrated prototype tactical functions and simulators that were developed independently by the HiPer-D organizations [Naval Surface Warfare Center Dahlgren Division (NSWCDD) and Johns Hopkins University Applied Physics Laboratory (JHU/APL)]. These were hosted on a layer of commercial off-the-shelf (COTS) software, using the full set of DARPA distributed computing technologies: the multi-node Touchstone Sigma (Paragon) computer, the Mach operating system kernel and, the Isis distributed communications toolkit.

b. HiPer-D I1 identified essential pieces of technology, not available as COTS or high performance computing technology, that would enable the initial building of high performance weapon systems. It identified the following technological requisites:

(1) A preemptive operating system with deterministic behavior that ran on high volume, low cost, standard workstation and server platforms (not the low volume niche market single board computers).

(2) Operating system instrumentation.

(3) Communication enhancements (simultaneous higher throughput combined with lower latency communications for small message sizes).

(4) System level instrumentation that could associate tactical application processing with resource consumption metrics from operating system instrumentation.

(5) Responsive fault tolerance mechanisms.

c. The I1demonstration met all of the stated program objectives. It validated the use of distributed computing technologies in future combat system design planning. The HiPer-D Integrated Demonstration 1 (I1) Report, published in September 1994, provided performance feedback to both the Aegis and DARPA communities on DARPA technologies in a simulated tactical environment. The report also recommended technology improvements, briefly assessed the process of developing combat system functions in a distributed environment and identified areas for further development.

d. The I1 demonstration, and subsequent publication of the report, ended Phase 1 of the HiPer-D Program.[1]

1.2 HiPer-D Phase 2 – DARPA/COTS Technology and Critical Issues Evaluation

a. In February 1994, HiPer-D Phase 2 began when programmatic lead and funding responsibility were transferred from DARPA to Aegis. Thus, the emphasis of the program correspondingly shifted. The HiPer-D Technical Management Team (TMT) was tasked by the Aegis chairman to focus on critical risks that had to be mitigated to ensure transition of distributed computing technologies into the Aegis Baseline 7 development program. Based on this guidance, the concept of an evolving engineering testbed was developed. Specific focus for HiPer-D Phase 2 included the following:

(1) Targeting specific baseline objectives.

(2) Ensuring HiPer-D demonstrations are Critical Issues driven.

(3) Building upon lessons learned and experience gained from HiPer-D Phase 1.

(4) Basing future efforts and demonstrations on re-engineered Aegis code.

(5) Using COTS products and open architectures where practical.

(6) Evaluating recommended improvements to DARPA technologies when ready.

b. The vision of the HiPer-D testbed was to permit collection and evaluation of engineering-quality data on new technologies and distributed system design concepts as they

[1] For further details see the HiPer-D Integrated Demonstration One (I1) Report, 6 September 1994

related to both future Aegis baselines and future surface combatants. This approach, and its evolution to support future system engineering development, would comprise a continuously evolving distributed computing testbed. The results of successive demonstrations of the testbed would feed into the various Aegis baseline development (forward fit and backfit) efforts. The base architecture serving as the departure point for HiPer-D Phase 2 testbed planning was Aegis Baseline 7 Phase 1 and follow-ons.

c. Since HiPer-D Phase 2 was driven by critical-issues, each of the HiPer-D technical organizations compiled a list of specific areas that it believed should be addressed in planning future distributed testbed demonstrations. The TMT consolidated this data into a Critical Issues List, and structured the information into five major subdivisions:

(1) Requirements and Architecture

(2) Combat System Functionality

(3) Tactical Support Layer

(4) Integrated Service Layer

(5) Process

d. The TMT then developed specific demonstration activities to address each of these areas. In order to target specific Aegis baseline objectives, demonstration activities were arranged, in conjunction with the Aegis programmatic schedules, to develop the HiPer-D distributed demonstration definitions and schedules. Table 1.2-1 details the consolidated information with results arranged into the above five major subdivisions.[2]

e. The testbed activities are centered on a series of formal demonstrations, T1 through T3. Each demonstration is composed of two parts, integrated demonstration and stand-alone demonstrations. The evolving integrated demonstration builds on previous milestones, and successively combines increasing Aegis Weapon System (AWS) functionality with the latest distributed computing techniques and technologies. The second element of each testbed demonstration includes a series of stand-alone experiments and demonstrations. The nature of the experiments and the components used vary as they are directed toward specific computing issues at a particular stage of the program.

[2] For comprehensive description of each critical issue see Appendix A, HiPer-D Phase 2 Engineering Test and Demonstration Plan, 1 November 1994

Table 1.2-1 HiPer-D Phase 2 Critical Item List

1. Architecture	• Architecture and System Partitioning Studies
2. Combat System Functionality	• Standard Missile Engagement Path (SPY, C&D, WCS) • Other AWS Tactical Functions as Required • Non-AWS Elements
3. Tactical Support Layer	• Track File Management • Track File • Doctrine Processing • Open Display / Submode Design • Navigation / Gyro Data Distribution • Parallelization
4. Integrated Service Layer	• Communications Protocol • Networks • Real-time Operating Systems • Fault Tolerance • Open System Issues, e.g., Client / Server Model • Distributed System Control and Resource Management • Time Management • Security
5. Process	• System Test • Re-engineering Legacy Code • Tool Initiatives • Language Issues: Ada, Ada 9X & Annexes, C, C++ • Baseline Strategy (Forward and Backfit)

1.2.1 HiPer-D Phase 2 Engineering Testbed One (T1) Demonstration

a. T1, on 15 May 1995, was the first formal testbed demonstration of the HiPer-D Phase 2 Program. Results were unprecedented, in that it proved that COTS could support the engineering development of large scale, complex, distributed computing-based systems. It addressed several shortfalls identified in I1, and achieved a significant increase in tactical track capacity.

b. Communications were addressed by moving from the Intel Paragon multi-computer and employing more mainstream COTS fiber distributed data interface (FDDI) network-based elements. Application of the COTS network was enhanced by a demonstration of an NSWCDD-defined, highly fault-tolerant FDDI network configuration with fault recovery times of 100 milliseconds. The Aegis SM-2 Auto-Special engagement function was developed and integrated into this test. Significantly, the HiPer-D track capacity was increased an order of magnitude from 100 (I1) to 1,000 tracks. This dramatic increase was due to locating, and resolving, the "track pipeline" bottlenecks over a period of several months.[3]

[3] For details concerning T1, see HiPer-D Engineering Testbed One (T1) Report, 17 November 1995

1.2.2 HiPer-D Phase 2 Engineering Testbed Two (T2) Demonstration

a. In October 1995, the second formal testbed demonstration of the HiPer-D Phase 2 Program, Engineering Testbed Two (T2), occurred. It consisted of the integrated demonstration, built on the previous T1 Demonstration, and stand-alone investigation of IP Multicast. The integrated demonstration included Alpha 3000s, with the Digital Unix 3.2B operating system, enabling an asynchronous transfer mode (ATM) switched network; and, the successful implementation of a hybrid network environment of FDDI and ATM. The tactical scenario sequence, with two Auto-SM doctrines established and Auto-Special doctrine activated, involved multi-sensor track initiations in the doctrine areas; appropriate engagements; random "no kill" assessments and re-engagements; subsequent ramping up of background tracks; faulting of servers; faulting of clients; and continued multi-doctrine engagements.

b. A new "peer client" model was developed and demonstrated for load sharing and fault tolerance. A new performance visualization tool based on the Isis Monitor of Performance (IMP) was developed to instrument and display peer client performance. Together, the Peer-Client model, used for Auto-SM, and the Radar Track Data Server (RTDS) design with replicated servers, enabled processing to be added dynamically in support of fault tolerance and load balancing. Application instrumentation (JEWEL) and the Peer-Client IMP X-Window display provided real-time distributed system performance monitoring and Go Plot provide post-exercise detailed analysis capability.

c. Significantly, the track entry rate in this demonstration was double that of the T1 Demo for the first 700 tracks entered. Subsequently, a capacity of more than 1200 tracks was exhibited. The T2 Demo was accomplished using Aegis-originated code. This included radar tracks from SPY and Surface Operations. The SPY Radar Control was derived from Lockheed Martin's SPY Control Loop; and Surface Ops was derived from Aegis Baseline 4 code. Finally, the message flow within the Auto-Special SM 2 engagement path was enhanced to reflect that of the current Aegis Weapon System.[4]

1.2.3 HiPer-D Phase 2 Engineering Testbed Two A (T2A) Demonstration

a. The HiPer-D Phase 2 T2A was held in December 1996. The stand-alone tests involved several efforts, among them were network tests on the Fiber Data and Distribution Interface (FDDI); Asynchronous Transfer Mode (ATM); Ethernet and Myrinet networks; Transport Control Protocol (TCP); User Datagram Protocol (UDP) throughput; and, bandwidth and latency for different platforms. Time synchronization tests using TCP were performed, and high data rate (gyro distribution) tests were also conducted.

b. The T2A demonstration successfully distributed gyro data over commercial network technology at 400 Hz, using a Gyro Data Converter simulator (GDC sim), with no degradation of tactical performance at (extremely high) track loads. Dynamic resource management was demonstrated through tactical applications automatically recognizing and reconfiguring after a

[4] For details concerning T2, see HiPer-D Engineering Testbed Two (T2) Report, 30 August 1996

system fault. Auto-SM was re-allocated to another homogeneous component based on CPU resource usage. Resource management functionality was added with a one-button start-up feature that brought all combat system functions on-line automatically, versus manually starting each. The Common Display Kernel (CDK) was successfully integrated into the HiPer-D testbed using the data broker concept. The Sensor Rate Server (SRS) demonstrated graceful degradation under severe system overload conditions by maintaining high priority tracks at the requested update rates, while reducing update rates for lower priority tracks.

c. By parallelizing more, and providing scalability through peer clients, effective track capacity increased to greater than 2,000 tracks in T2A. A track capacity of 2200-2400 tracks was achieved without SRS intervention, and a total of 4900 tracks was supported with SRS intervention.

d. T2A also compared network technology using Aegis gyro data requirements; specifically, FDDI versus ATM versus Myrinet, to determine which of these would provide the most bandwidth, scalability, and particularly, the lowest latency.[5]

1.2.4 HiPer-D Phase 2 Engineering Testbed Three (T3) Demonstration

a. All of the software used in the T3 testbed was commercial off the shelf (COTS), and the hardware was a heterogeneous mix composed of work stations from four vendors, each using the vendor-supplied operating system, and three networks.

b. One of the major goals for T3 was to expand the scope of the testbed beyond shipboard Anti-Air Warfare (AAW). To accomplish this, C^4ISR components, the Joint Maritime Combat Information System (JMCIS) and Advanced Planning and Power Projection and Execution (APPEX) were interfaced to the Aegis AAW subsystem by data brokers. Data brokers are small computer programs that, essentially, translate data from a transmitting component into a format that can be processed by the receiving component. This is a concept that could be used to integrate legacy systems into a new architecture. The Track Data Broker and the C^3I data broker were two new components developed for T3. During the demonstration, JMCIS provided over the horizon (OTH) track information throughout the system, received real-time AAW tracks, and provided track data to APPEX.

c. Another important goal was to demonstrate the ability to respond to a Naval Surface Fire Support (NSFS) call for fire. To accomplish this goal, a scenario was scripted that included landing Marines ashore, having multiple NSFS calls for fire, some with 5" gun engagements and some with surface to surface missile engagements.

d. The HiPer-D Resource Manager (RM) was used to start non-HiPer-D components automatically and remotely at the beginning of the demonstration. Resource Management for T3

[5]See HiPer-D Engineering Testbed Two A (T2A) Report, dated 16 Dec 97, for peer client architecture and implementation details

had two dimensions. The first was system recovery in the event of a fault. The other was the adding of resources when the system became overloaded.

e. A video from a Predator Unmanned Aerial Vehicle (UAV) was run during the T3 demonstration. It was displayed on the Anti-Surface Warfare (ASUW) Coordinator's workstation as well as Tactical Air Operations (TAO) workstation. The goal was to determine the cost, in CPU usage and network bandwidth, to digitize, compress and send out the signal at 30 frames per second.[6]

1.3 Demo 98 Objectives

a. The technical objectives for Demo 98 included:

(1) To expand the land attack capability by integrating the Advanced Tomahawk Weapons Control System (ATWCS) Launch Control Real Time (LCRT) group into the testbed. This required the porting of the LCRT program from HP743RT to Sun Solaris and still meeting the ATWCS LCRT real time requirements. Also the inertial navigation data and the interface with the vertical launch system (simulator) were to be over the tactical network and not point-to-point NTDS interfaces as in the operational system. ATWCS was to use the Network Time Protocol (NTP) over the tactical network for clock synchronization. ATWCS was also to integrate with the HiPer-D Resource Manager for both start-up and fault tolerance ("hot restart" of LCRT). Finally, the ATWCS LCRT was to be instrumented with jewel to be able to assess performance in near real time during the demonstration.

(2) To enhance the Naval Surface Fire Support call for fire from voice communication to a digital capability. In T3 a voice communication (simulated) call for fire was demonstrated. For Demo 98 the objective was to have a remote Forward Observer/Forward Air Controller (FO/FAC) issue a digital call for fire over the network to ownship. Ownship was to respond with a gun engagement and the appropriate tacfire messages (shot, splash, spot adjust, etc.) were to be exchanged between ownship and the FO/FAC.

(3) To complete the development and integration of a fully fault tolerant, scalable AAW path through the system. This involves the development of a fault tolerant Engagement Server function. This function maintains all system state data or status regarding ongoing or requested engagements by guns or missiles. The preservation of this engagement state data during faults, failures, and recoveries was to be demonstrated. The impact of these faults, failures, and recoveries during an engagement on the SPY Auto-Special timeline was also to be examined. A fault tolerant Advanced Track Correlator and Filter (ATCF) is also an objective for this demonstration. The two processes, Track Control and Track Processor, are to be replicated for fault tolerance.

(4) To enhance the Resource Management capability of the system. The objective included increased functionality (the scaleup of weapons doctrine components as track load increased based on a Quality of Service track review time specification), reduced fault recovery

[6] See HiPer-D Engineering Testbed Three (T3) Report, dated 31 Dec 1998, for further details.

time, (programs that faulted and restarted in much less than one second), and the starting and restarting (due to program faults) of non-HiPer-D components such as ATWCS.

(5) To further the evolution toward Navy open systems. This included the installation, test, and use of the open network standard IP Multicast capability over three networks (ATM, FDDI, and Ethernet). IP Multicast allows the message sender to transmit one message that is received by many receivers.

(6) To begin assessing the feasibility of using COTS-based distributed object computing technologies for the Navy. This included the integration of Common Object Request Broker Architecture (CORBA) middleware technology in the Track Number Server (TNS) and the Doctrine/Plan Server components.

(7) To evolve toward long term middleware. This included the integration of various commercial products CORBA-based, and NDDS (a publish/subscribe communications package), and the integration of SPREAD group communications in the ATCF component.

(8) To expand the simulation capability of the testbed by adding a physics-based, DIS compatible wraparound simulation capability. The physics based capability would allow more operationally oriented and operationally valid scenarios to be a part of future demonstrations. The DIS capability would provide the basis for integrating readily with other simulators and integrating geographically dispersed laboratories into the HiPer-D testbed.

(9) To include the real time, high data rate delivery of navigation/gyro data. The ATWCS LCRT component requires a 16 Hz navigation data input which was to be provided over the ATM network.

b. The remainder of the report describes the demonstration components and the configuration, the demonstration scenario, how these objectives were met, and lessons learned from the effort.

2.0 STAND-ALONE ENGINEERING TESTS

Stand-alone Engineering Tests provide data concerning the use and applicability of a technology or product to the integrated demonstration. These tests are aimed at "breaking" technologies and providing insight in the maturity, robustness, and performance under heavy loads. Three Stand-alone Engineering tests are summarized in the following subparagraphs with the details provided in appendices.

2.1 Evaluating the Performance of Multicast Communications

In the distributed shipboard environment of interest to the U. S. Navy, there is an increasing interest in the use of multicast communications to reduce bandwidth consumption and to reduce latencies. The bandwidth required to transmit large volumes of information (e.g., track files, maps, etc.) to multiple receivers could potentially be reduced significantly by the use of multicast data transmission. Many types of real-time shipboard data, such as navigational and gyro data, need to be distributed to a large number of hosts. The distribution of this type of data might also benefit from the reduced latency possible using multicast techniques instead of sequential unicast transmission. Before multicast communications can be used in this environment, however, a characterization of its performance must be made. Appendix A proposes a number of metrics, and data collection and analysis techniques for assessing multicast communications performance. Of particular significance is a metric that correlates reception of message and shows promise in analyzing topology-related problems. While the concepts presented in Appendix A are applicable to the general forms of multicast, the appendix specifically focuses on the use of IP Multicast in an internal shipboard environment. The MCAST Tool Suite (MTS), which uses the metrics and data collection techniques presented, is described. The results of applying this toolset to simulate and instrument several IP Multicast-based application scenarios are presented. See Appendix A for details.

2.2 Data Distribution Experiment

The data distribution problem domain for command and control systems can be divided into two general categories: control data and streaming data. Complex data ordering, low volume, reliable delivery, and deterministic latency often characterize control data. Examples of control data for the Aegis Combat System could be Auto-Special and Doctrine data. Streaming data usually has limited data order dependencies, high volume, requires a stable frequency and inter-arrival, and does not require reliable delivery. A message or possibly several messages could be missed depending on the frequency and message type, and when the next message is received the system requirements still would be satisfied. Examples of streaming data could be gyro and track file update data. Obviously not all the message types used in the Aegis Combat System fall neatly into one of these two categories. SPY data, for example, could be considered high volume and yet require a stringent deterministic latency. The future Aegis Baseline 7 Phase 1 timing requirements are currently being reviewed on a message type basis. Tradeoffs, such as deterministic latency versus reliable delivery, are being evaluated. These future Aegis baselines will use distributed processing architectures and distributed applications that employ commercial off the shelf hardware and software to the greatest extent possible. Software companies have developed, and continue to develop, a broad variety of commercial middleware products to help

system developers implement distributed applications in the data distribution arena. One group of products that appears to meet the data streaming requirements is publish/subscribe products. Appendix B documents the results of the evaluation of two publish/subscribe products.

2.3 Windows NT Investigations

In June 1998, the Navy's Chief Information Officer (CIO) released his Information Technology Standards Guidelines (ITSG). The guidelines recognized the growing presence of Windows NT, as well as its possible application to the Navy's requirements for a powerful operating system. The ITSG indicated an organization-wide shift towards NT over the next few years, encompassing systems such as ashore and on-ship installations. The question now was not "Should we use NT?" but " What do we need to do to make NT work the way we want?" The Windows NT investigations documented in Appendix C are the beginning of this effort whose goal is to answer the latter question for HiPer-D.

3.0 ADVANCED COMPUTING TESTBED DEMO 98 INTEGRATED DEMONSTRATION DESCRIPTION

a. The Demonstration 98 milestone event was held on September 29, 1998 in the System Control Laboratory, Building 1500, at NSWCDD. Among the attendees were CDR Stevenson of PMS 400 and Dr. Gary Koob of DARPA. The demonstration was imminently successful in that all the following technical objectives were met:

(1) To expand the land attack capability by integrating the Advanced Tomahawk Weapons Control System (ATWCS) Launch Control Real Time (LCRT) group into the HiPer-D testbed.

(2) To enhance the NSFS call for fire from voice communications to a digital capability.

(3) To complete the development and integration of a fully fault tolerant, scalable AAW path through the system. This involved the development of a fault tolerant Engagement Server and a fault tolerant Track Correlator and Filter (TCF).

(4) To enhance the resource management capability of the system. This included increased functionality, reduced fault recovery times, and starting and re-starting of non-HiPer-D components, such as ATWCS.

(5) To further the evolution toward Navy open systems. This included the expansion of the ATM network and the addition of IP multicast capability over all three networks (ATM, FDDI, Ethernet).

(6) To begin assessing the feasibility of using COTS-based distributed object computing technologies for the Navy. This included the integration of CORBA in the Track Number Server (TNS) and the Plan Server components.

(7) To evolve toward long term middleware. This included the integration of commercial products, CORBA-based and NDDS, and the integration of SPREAD group communication in the Advanced Track Correlator and Filter (ATCF).

(8) To expand the simulation capability of the testbed by adding a physics-based, DIS compatible wrap-around simulation capability. The DIS capability would provide the basis for integrating geographically dispersed laboratories into the HiPer-D testbed.

(9) To include navigation/gyro data distribution. The ATWCS LCRT component requires a 16 Hz navigation data input which was delivered over the ATM network.

b. The Standard Missile engagement path through the AWS, as shown in Figure 3.0-1, was initially chosen for prototype implementation in the test bed prior to the I1 demonstration in 1994. The standard missile path provides a high performance challenge with well-defined timing requirements for experimental validation purposes. The tactical function in this AAW path is

essentially the same as in the AN/UYK-43 based AWS computer programs. Modifications were made that kept the prototype unclassified and do not affect results.

c. Each successive demonstration since has added AAW capability to this engagement path and with the T3 demonstration and this demonstration the test bed functionality has been expanded into C4ISR and land attack areas as shown in the Demo 98 Block Diagram, Figure 3.0-2.

(1) The gray boxes are the wraparound simulation components that provide the scenario inputs, control the scenario timing, and process the outputs of the tactical components. The Environment Simulation (Env Sim) provides the scenario inputs and controls the scenario timing. The MFAR Sim simulates a physics based multi-function array radar. The ID Sim simulates the ID output of an IFF system. The OTH simulator provides OTH sensor data to JMCIS. The FO/FAC and RDDL components simulate a remote digital CFF request from a Forward Observer. The Engagement Planning Sim (EP Sim) and the Mission Data Sim (MD Sim) provide mission planning inputs into the ATWCS LCRT group. The Vertical Launching System Sim (VLS Sim), the Missile Sim, and the Flight Sim receive the ATWCS mission plans and missile alignment data, and fly the ATWCS missiles to their assigned targets. The Navigation simulator (Nav Sim) and the gyro data simulator (Gyro Sim) provide nav/gyro data to the system. The Weapons Control system sim (WCS Sim) simulates the SM-2 missile system and the Naval Surface Fire support simulation (NSFS Sim) simulates a 5" gun system. Kined is a bulk track load generator used to input thousands of tracks into the system to demonstrate system performance under heavy load.

(2) The AAW components are shown in blue. The Advanced Track Correlation and Filtering (ATCF) components receive AAW tracks from the MFAR Broker and Kined. When updates or new track data are received they are passed to the Radar Track Data Server (RTDS) and the server provides AAW track data to all system components or clients that need the data. The weapon doctrines (Semi-Auto, Auto-Sim, and Auto-Special) receive and process the track data and send engagement requests to the Engagement Server for those tracks determined to be threats based on the currently activated weapons doctrine. Manual Engage Control provides operator selected, or operator approved engagement requests to the Engagement Server for processing. The Engagement Server validates the engagement request and forwards the engagement order to either WCS Sim for missiles or NSFS Sim for gun engagement. Two AAW operator positions were utilized in Demo 98 using a geographical tactical picture display that showed the AAW and OTH tracks and activated weapon doctrines. These two positions were the Anti-Air Warfare Coordinator (AAWC) and the Surface Warfare Coordinator (SUWC). The double boxed components are fault tolerant and/or scalable (ATCF, RTDS, Semi-Auto, Auto-SM, Auto-Special, Engagement Server).

(3) The green components represent the land attack capability in the demo. The Launch Control Executive (LC Exec) and the Launch Control Real Time (LCRT) are ATWCS components. The Tacfire, C3I Broker, and Advanced Battle Management and Execution (ABMX) are part of the digital CFF capability.

(4) The brown components are the C4I processes. JMCIS is a SPAWAR C4I product that provides a C4I picture of current operations. The Aegis Air Correlator Tracker (AACT) provides the interface between the C4I subsystem and the AAW subsystem allowing the passing of OTH and AAW track data between the two. The OTH Data Server provides the OTH track data to all system processes or clients that need OTH track data (ex., AAW operator displays).

(5) The AAW components are described in detail next, followed by the ATWCS land attack component and the JMCIS C4ISR components. Next the Resource Management Component is described and then the simulation components. Following the component description, the demonstration scenarios and major events are presented, then lessons learned from the effort.

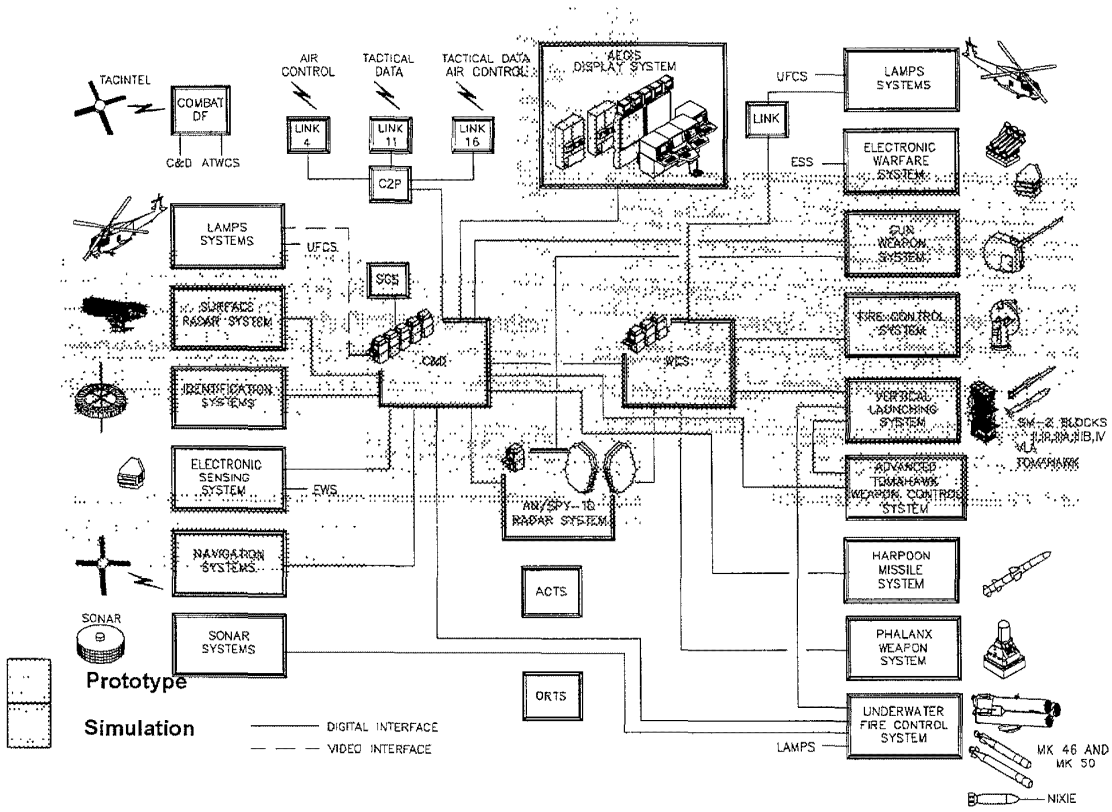


Figure 3.0-1 Aegis Weapon System with HiPer-D Overlay

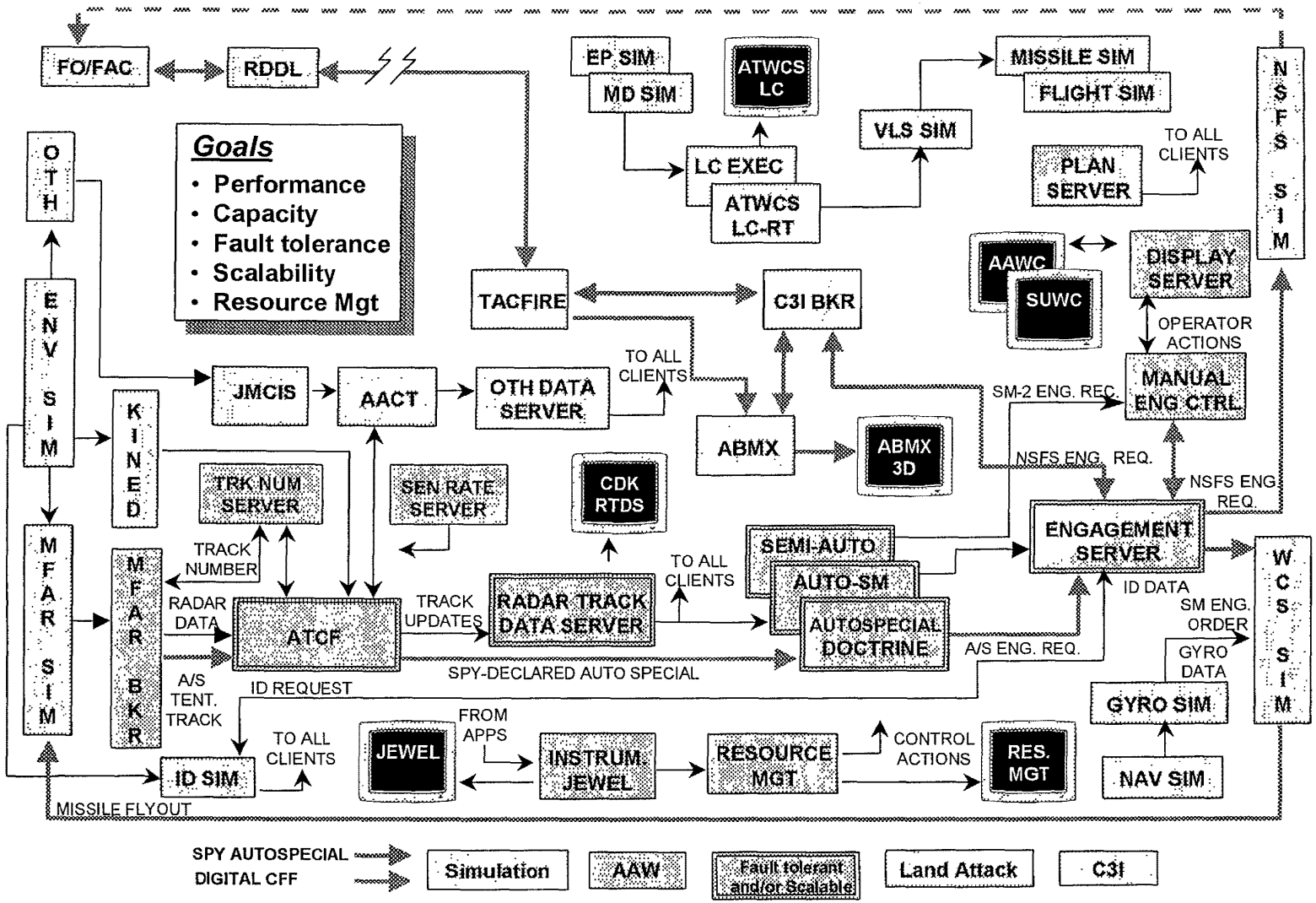


Figure 3.0-2 Demo 98 Block Diagram

3.1 AAW Subsystem Functional Description

a. The AAW subsystem components are highlighted in Figure 3.1-1. These components interface and receive sensor inputs from the sensor simulators, process the data into a track report, distribute the track information to all client components that need access, process and evaluate the track data for threat evaluation, and engage those tracks deemed hostile according to the specified ship doctrine.

b. The major enhancements and additions to the AAW subsystem for Demo 98 include:

(1) Redesign of the track correlation and filtering components for increased capacity and fault tolerance.

(2) Incorporation of CORBA technology in the Doctrine/Plan server and the Track Number Server.

(3) Redesign of the state data intensive Engagement Server to provide a primary/shadow fault tolerance capability.

c. A description of all the AAW subsystem components follows with additional details describing the major enhancements and additions.

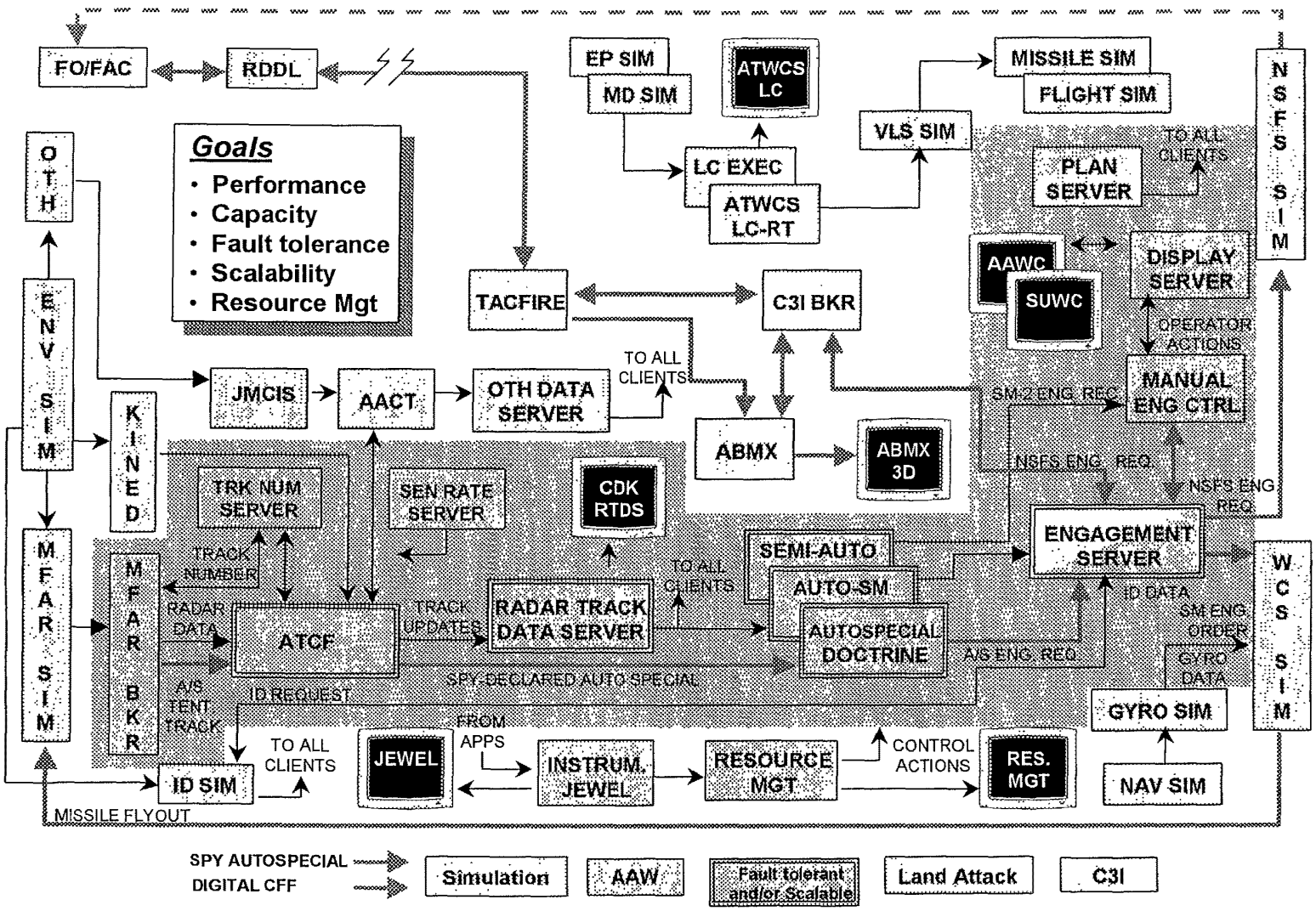


Figure 3.1-1 Demo 98 Block Diagram with AAW Components Highlighted

3.1.1 Advanced Track Correlation and Filtering (ATCF)

a. The track correlation and filtering elements of HiPer-D originated as a port of the "Milestone 90" Cooperative Engagement Processor (CEP) correlation and filtering capability to HiPer-D. This became known as HiPer-D Correlation and Tracking (HCT). It was used in the initial HiPer-D integrated demonstration, I1. The next generation was aimed at making the correlation and filtering more directly relevant to Aegis. This build, called Track Correlation and Filter (TCF), implemented correlation and filtering algorithms from the Aegis C&D PPS, and was used in the T1, T2, T2A, and T3 HiPer-D demonstrations. Program focus during the development of HCT and TCF was on issues of track distribution. In this environment, efforts were not expended to make either component fault-tolerant or scalable. See Appendix D for details of the history of HiPer-D Track Correlation and Filtering processing.

b. In T2A and T3 the TCF's total update rate of approximately 2300 updates per second began to limit the throughput of the overall system significantly. This prompted the development of the Advanced Track Correlator and Filter (ATCF) for the 1998 demonstration. ATCF would be designed to be both scalable and fault-tolerant.

3.1.1.1 ATCF Overview

a. The Advanced TCF addressed several issues to improve upon the original TCF. Some of these improvements were not fully realized in 1998, but the majority of the necessary infrastructure to support them is in place. In particular, neither correlation nor filtering is implemented yet. To date, the ATCF effort has focused on building the proper communications architecture. The design looks similar to the original TCF but has a shift in functional allocation that better addresses scalability. The two components have been renamed Track Controller and Track Processor (see Figure 3.1.1.1-1). The ATCF is fully fault-tolerant and can distribute the load across a dynamically changing number of Track Processors.

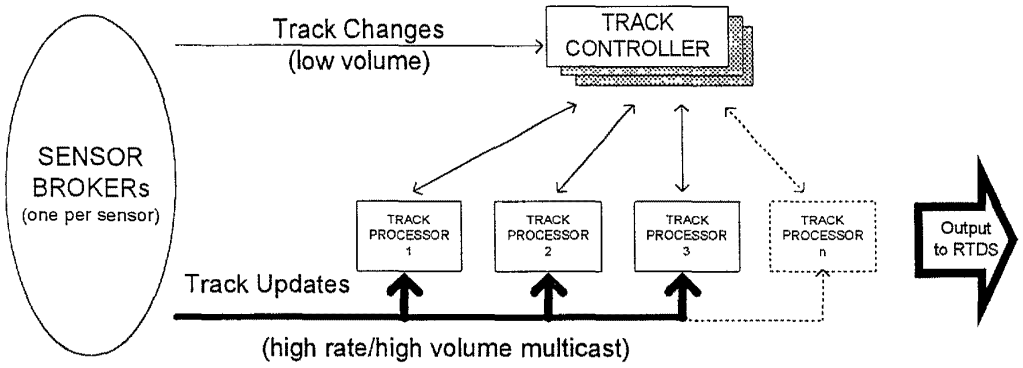


Figure 3.1.1.1-1 Advanced TCF

b. The major issue with TCF was the complexity of scaling the Track Init function. In ATCF, the primary computation involved in correlation has been shifted out to the Track Processors, with the Track Controller retaining a role as "coordinator". Essentially, this shifts all major processing to the Track Processors, thus reducing the need to scale the Track Controller. Fault tolerance is achieved in the controller by providing a hot spare, and in the Track Processors

by allowing dynamic reallocation of track responsibilities. Track Processors split the processing load by track. Each processor has a collection of tracks for which it is responsible. A new Track Processor receives a portion of each existing Track Processor's track load, and a failing Track Processor's responsibilities are redistributed over the remaining nodes.

3.1.1.1.1 Standard Message Format

a. The first step in building ATCF was to specify the input message formats. Each sensor provides an independent stream of data, with slightly different characteristics. A single message format called "sensor 3D" was defined for use within ATCF. This defines the necessary components of data that ATCF expects to receive from a contributing sensor. The introduction of brokers allows the input messages from a sensor to be converted to the standard sensor three-dimensional (sensor 3D) format. All sensors (via their brokers) submit sensor 3D messages to ATCF. As a result, all messages from all sensors can now be processed by the same logic within ATCF. The use of brokers enables a new sensor to be added to a system without having to modify the ATCF to account for the new sensor's message type. A broker simply converts the sensor's unique message format to the sensor 3D format. The convenience and reduced complexity of handling a single message type far outweighs the effort of developing any new brokers. A broker was developed for the KINED[7] sensor/simulator to demonstrate a sample implementation. This adapts the KINED output to the new ATCF sensor 3D input format. No changes to KINED were required.

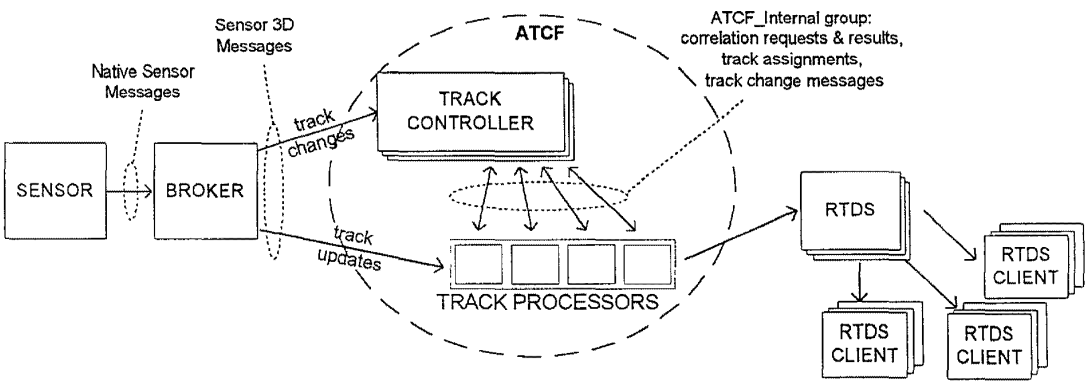


Figure 3.1.1.1.1-1 ATCF I/O

b. During a typical simulation run, KINED generates <u>all</u> track messages (new tracks, updates, and track drops). The addition of a KINED broker has increased the number of processes through which all traffic is routed. (Note the addition of a broker in Figures 3.1.1.1.1-1 and Figure 3.1.1.1.2-1.) This caused a slight increase in the end-to-end (KINED to RTDS client) latencies. Track change message latencies increased approximately 2 ms, and track update latencies increased approximately 10 ms.

3.1.1.1.2 MFAR Broker

[7] KINED is a simulator created to model multiple sensors and drive the system with input data.

a. A second broker was created for adapting the MFAR simulator's[8] output to the new ATCF (see Figure 3.1.1.1.2-1). The MFAR broker provides a bridge from one communications mechanism (CSSEnet) to another (process group communications). MFAR broker accepts new track, update, and drop track messages from the MFAR simulator. These messages are converted to the Sensor 3D format and sent on to ATCF. The broker also receives Auto-Special doctrine messages. The Doctrine/Plan Server sends doctrine messages in the Auto-Special communications group. If MFAR broker receives an Auto-Special doctrine, new track messages from the MFAR simulator will be compared against the doctrine parameters. Any new tracks meeting the criteria are flagged as Auto-Special when sent to ATCF. This also triggers special logic in the broker to flag the first update received for each Auto-Special track. This first update must be issued as an Auto-Special resolution.

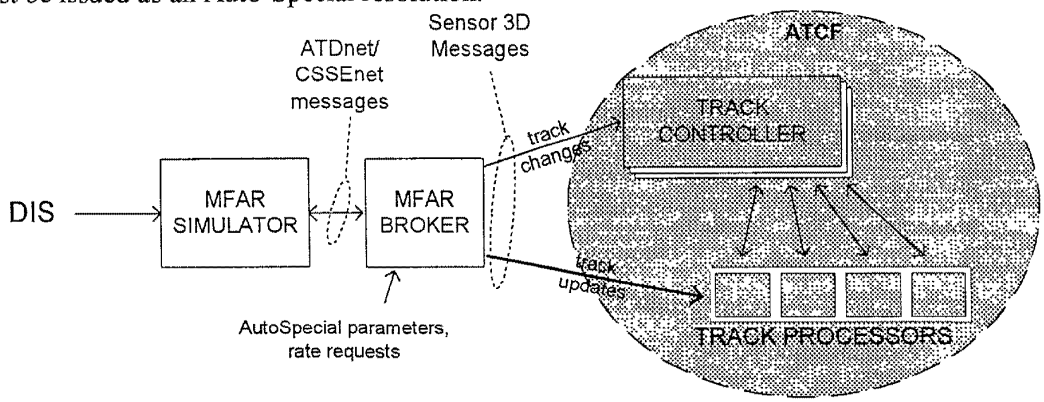


Figure 3.1.1.1.2-1 MFAR Broker

b. The MFAR simulator supports rate control for individual tracks. A track's update rate is the number of times an update report is issued for that track during a unit of time. A separate process within HiPer-D, the Sensor Rate Server (SRS), determines when a rate for a particular track should be adjusted. The MFAR broker receives these requests and forwards them to the MFAR simulator. The MFAR simulator adjusts the rate at which updates are passed to the MFAR broker.

c. Multiple brokers can coexist and operate concurrently. Brokers may join and/or leave a running system without disrupting the ATCF.

3.1.1.1.3 IP Multicast Communications

a. The next step in building ATCF was to develop a messaging layer to allow the brokers to deliver updates to the Track Processors with maximum efficiency. First a communications protocol selection was required. IP Multicast is inherently unreliable, and extra steps are needed to verify that all messages get delivered to all listeners. Process group communications libraries such as ISIS or Spread implement reliability and ordering at the expense of increased latency in message arrival. From a system perspective, it is more important to maintain a constant flow of timely updates than to expend extra effort checking that each and

[8] The MFAR Simulator is a component of NSWCDD's CSSE simulation environment. It takes input from DIS and filters it using physical characteristics to provide reports that more accurately repreent the returns seen by a radar.

every update gets delivered, in order. Given this, IP multicast was selected as an experimental candidate. The question explored by this experiment was could an ATCF design use the lower quality protocol and still provide high speed reliable service? The use of IP multicast was significant because it is the first use of unreliable communications in a critical path within HiPer-D.

b. The ATCF design used timestamps in the updates to be sure that time ordering of reports is not grossly violated. If two updates for a single logical track arrive out of order, the older one is discarded. Because the flow of updates is constant, allowing a small percentage to be dropped should not have a significant impact. The track change messages (new track messages and track drop messages) are still reliably delivered in process communications groups.

c. Performance statistics were gathered to aid in evaluating the use of multicast. The statistics being measured include the percentage of lost messages, the percentage of messages that arrive out of order, and the average latency of the messages being delivered. Testing was performed on Ethernet, FDDI, and ATM networks. The only behavioral difference displayed by each type of network was that each medium provided a different maximum total track capacity. Message loss and misordered delivery of messages was only observed upon nearing maximum system capacity. Message loss was quite rare on ATM networks.

d. Timed buffering was implemented to improve throughput. A timed buffer is a queue in which messages to be sent are placed. A maximum age, also known as a staleness value, for the messages in the queue is set. A separate thread of execution (separate from the sending thread) observes this buffer and sends the contents out at regular intervals as defined by the staleness value (typically 10-50 ms). The buffer is also sent when it is full. Without modifying the operating system, the minimum effective interval is approximately 10 ms.[9] Timed buffering causes multiple messages to be sent as one unit, resulting in fewer socket calls. This is more efficient than making a socket call for each message, and results in a much higher maximum throughput. The tunable parameters are the staleness interval and the size of the buffer. The latency for each message is directly related to this staleness value. End-to-end (KINED to RTDS client) latencies as low as 40 ms were observed when all staleness intervals were set to 10 ms and a moderately high number (several thousand) of tracks was injected into the system. Average total (end to end) latencies were in the range of 50 to 70 ms for a load of around 1000 tracks. As the load increases, these timed buffers begin to fill completely before the interval expires, causing more frequent sends to occur. This has the effect of pushing the total latency down as the load increases.

e. One point of interest regarding buffer size is that each network standard has a maximum transmission unit (MTU) size. For Ethernet, this is 1500 bytes. For FDDI, the MTU size is around 4096 bytes. For ATM, it is dependent on the network interface driver (the actual *packet*, or *cell*, size is 53 bytes). On the Solaris 2.6 workstations with ATM interfaces in the labs

[9] This was the case with single and multiprocessor workstations running Solaris 2.5, Solaris 2.6, and Digital's OSF/1 4.0. This is believed to be a side effect of the process scheduler allocating minimum time slices to each process before preempting them.

at NSWCDD, MTU size was around 9000 bytes. It is acceptable for an application to request to send a block of data larger than the MTU, but this will result in the message being fragmented into multiple units, or packets. The fragments must then be reassembled by the receiving entity's kernel. It was expected that the overhead of fragmentation handling would cause a decrease in performance, but this was not seen. In fact, using buffers larger than the MTU proved to be more efficient than using smaller ones. This may be due to the fragments being sent as back-to-back packets on the Ethernet and FDDI mediums, but this is purely conjecture. Another possible explanation is that as the blocking size of the data is increased, the number of socket calls needed decreases, for a constant amount of data (multiple blocks' worth). The time needed to pass a block of data up or down through the protocols in the network stack may be significant and independent of the size of the block of data.

f. The operating system also provides the ability to change the buffer size used within the kernel for socket operations. This is not the same buffer used by a communications library for timed buffering. Sizes between 4 Kb and 512 Kb were tried, but no effect on performance was observed. A separate system call is performed to verify that the request to change the socket buffer size was indeed being acknowledged by the operating system.

g. The multicast library does not ensure that messages get delivered (i.e., does not implement reliable delivery). An incrementing counter is placed in each message sent. The receiver simply remembers this counter value from the last message. If the next message received has a counter value greater than (the last counter + 1), this indicates that some messages were dropped. The exact number can be computed. The counter values used by the multicast library are only used as a means of gathering statistics on performance. Reordering and retransmit have not been implemented.

h. Each track update message contains a timestamp, which is placed there by the sensor or simulator, indicating the time of contact. It is important that updates for a single logical track are not processed out of order. It is acceptable that some updates may be discarded as a result. These updates would have been delivered across the network in the wrong order. The timestamp can be used to prevent processing an older update after a newer update. If a client were to receive an older update after a newer one, it might appear that a track had changed direction, or some other oddity. The ATCF only processes updates that are newer than the last received update for a given track.

i. The application-programming interface to the multicast library was modeled using Amalthea domain classes. Functions provided include: an initialize method, a callback registration method for message delivery, a group join method, a method to send messages, and a group leave method. Also, a method to enable and configure the timed buffering option was provided.

j. Message buffering was experimented within the *track change* process communications group. With buffering enabled, several thousand tracks could be injected into a system in a matter of seconds. However, the latency for all track change messages was increased by approximately 8 ms as a result. Without buffering the *track change* group, the time required

to feed several thousand tracks into the system increases dramatically, but keeps these latencies to a minimum. This is the preferred option, since low latency is required for AutoSpecial events.

k. As a passive form of flow control, a maximum latency threshold can be specified for messages received by a Track Processor. Presumably, if track updates are being received with "old" timestamps, then the network medium is likely to be congested. The appropriate course of action is to drop these aged reports, rather than propagate the condition by introducing more traffic (delivering an update to a track processor will result in another update message being sent to the RTDS). It is likely that the updates would be "aged" further before final delivery to RTDS's clients. Some clients might wish to impose maximum latency thresholds for the track updates. Therefore, it is in the best interest of the system to discard reports if they are too old. Track Processor currently supports only a static threshold.

l. The Track Processors use a process communications group to deliver all messages to the RTDS. Track change messages are combined with track reports in this group. This allows the Track Processors to guarantee that the RTDS will not be subjected to illegal sequences (e.g. receiving a track update before a new track message, or receiving a track update after a lost track message). The ordering logic in Track Processor maintains a recent history of dropped tracks, in case any reports arrive after the drop message. This prevents Track Processor from forwarding any reports after sending a lost track message.

m. All messages emanating from a single Track Processor must be processed by all RTDSs in the order they were sent (to ensure track update ordering). However, ordered streams from different Track Processors may be interspersed in any fashion. The design of ATCF supports independence between Track Processors (i.e., they do not communicate directly to coordinate their activities with one another).

3.1.1.1.4 ATCF Fault Tolerance

a. The Track Controller is implemented with a primary/shadow form of fault tolerance. Primary/shadow (sometimes called active/backup) means that redundant replicas of a process are maintained. This is accomplished by having all processes consume input data with only the designated (active) instance providing output. If the active instance fails, then one of the backup replicas can be promoted to the active status.

b. The Track Controller directs and coordinates the activities of the Track Processors with respect to fault tolerance. The ATCF uses all available Track Processors and has the ability to recover from a total loss of Track Processors. If all Track Processors should fail, there is a 90-sec window during which a new Track Processor may be started. No updates are delivered to RTDS clients during this time. If a new Track Processor is created within this time window, all tracks are assigned to this new Track Processor, and reports begin to flow again. If no new Track Processor becomes available after 90 sec, RTDS will issue drop track messages for all tracks that were being reported. If a new Track Processor was created after this event, and reports began to arrive at the RTDS again, RTDS would issue new track messages to the RTDS clients. The flow of updates would continue to the RTDS clients. The ATCF can even recover as long as at least one Track Controller remains.

3.1.1.1.5 Track Number Mapping

a. Track number mapping occurs when a process transforms the track identifiers provided in the input to a different set of identifiers for its output. Each sensor has its own method of assigning identifiers (usually a number that can be represented by 16 or 32 bits) to tracks. These identifiers are unique only within the domain of that one sensor. Therefore, in a system with multiple sensors, it is entirely possible that two sensors might use the same identifier for two logically different tracks. Each identifier would be unique to one sensor, but by coincidence two sensors may select the same identifier. This is the reason number mapping is performed. For each sensor, a mapping can be constructed to ensure that each logical track maps to its own unique system-wide identifier. This is accomplished by drawing all mapped output identifiers from a common pool of identifiers. The TNS was created for exactly this purpose.

b. Figure 3.1.1.1.5-1 illustrates the intended use of the TNS by various processes. Typically a broker (Label 1 of Figure 3.1.1.1.5-1) performs the mapping function for each sensor. This ensures that all sensors' track contributions can be uniquely identified by ATCF. An output identifier from a broker is called a sensor HTN (held track number). Track Controller allocates numbers to logical track entities (Label 2a of Figure 3.1.1.1.5-1). The ATCF output identifiers are called composite HTNs. If two tracks being reported by different sensors are correlated, they can be mapped to the same composite HTN. The sensor HTN to composite HTN mappings maintained by Track Controller get distributed to the Track Processors in the track assignments (Label 2b of Figure 3.1.1.1.5-1). Note that the identifier mappings created by Track Controller are applied to track change messages issued to the Track Processors, but the track update messages contain sensor HTN identifiers. Track Processor must transform the sensor HTNs to composite HTNs before updates are sent to RTDS.

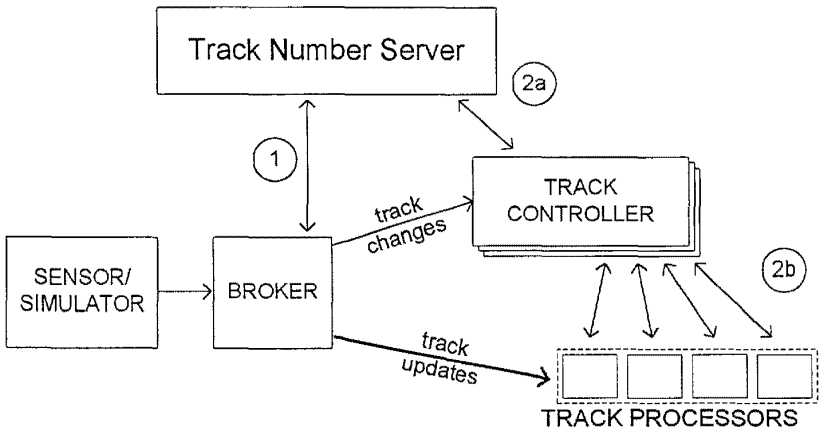


Figure 3.1.1.1.5-1 Track Number Change as a Result of Correlation

3.1.2 Air Engagement Control (AEC)

a. The AEC implements a portion of Aegis C&D and WCS functionality. It performs the processing necessary to evaluate potentially threatening air tracks and initiate engagement of

those tracks with SM-2 missiles. AEC evaluates air threats by comparing track data provided by the HCT against a priori defined criteria such as range, velocity, and identification. These criteria are referred to as doctrine. AEC engagement doctrine may be semi-automatic (operator in the loop), automatic, and/or sensor-initiated automatic doctrine, also known as Semi-Auto, Auto-SM, and Auto-Special doctrine, respectively. An engagement server mediates engagement requests from the doctrine processing applications and performs validity checks on the engagements. It also provides the interface with the weapons control system to initiate engagements and monitor their progress. Supporting applications include an Aegis-like display capability with submode and alert routing support as well as tactical and character read-out (CRO) displays, a manual engagement support application, a doctrine server that acts as a repository of doctrine criteria, and a variety of simulators and instrumentation tools.

b. The AEC uses an open systems approach for its code base. This approach simplifies the process of adding tactical functionality. The architecture relies on client/server design principles, software layering, and data driven design. Libraries of reusable and common services are used as building blocks upon which application unique functionality can be added. These common services facilitate interfacing with other applications and include domain specific services that abstract inter-application interface details including messaging and data marshalling as well as data caching. Examples include track data management services, engagement client services, and display services. A group-based communications API provides an abstraction for group communications that is currently implemented using Ensemble but is not dependent upon a particular implementation. The common services also include more general-purpose functionality such as UDP and TCP communications service, queue management, string manipulation, and semaphores.

3.1.2.1 AEC Component Summary

Table 3.1.2.1-1 summarizes the AEC components. The major enhancements to the AEC components for Demo 98 were the addition of Common Object Request Broker Architecture (CORBA) to the Doctrine Plan Server and the addition of fault tolerant capability to the Engagement Server.

Table 3.1.2.1-1 AEC Component Descriptions

ELEMENT	*FUNCTION*
AEC Display	• General Purpose Tactical Display Capability • X-Windows Based Map and PPI Using ADS Map Database • CRO, Submodes, Alerts, Tutorials, Close Control, FAB / VAB, and Fixed Map Control Support
Display State Data Server	• Alert Routing • Submode Mediation / Control • Display State Repository in Support of Open Systems Architecture
Manual Engage Control	• Engagement Client • Manual Engagement Initiation • Operator Interface to Engagement Status
Semi-Auto-SM	• Semi-Auto-SM Doctrine Qualification • Fault Tolerant/Load Sharing
Auto-SM	• Auto-SM Doctrine Qualification • Fault Tolerant/Load Sharing
Auto-Special	• SPY Auto-Special Doctrine Qualification • AEC Auto-Special Doctrine Qualification • Fault Tolerant/Load Sharing and Active Replication
Engagement Server	• Engagement Control • Engagement Monitoring • Tactical / WCS Sim Interface • Target Engagement Status Server • Primary/Shadow Fault Tolerance Capability
Doctrine/Plan Server	• Provides Doctrine Statements/Updates Server • CORBA and Isis Interface Support

3.1.2.2 AEC Display

The tactical display capability is provided by the AEC Display Program and the AEC State Data Server, as well as a reusable layer of code that facilitates use of the display capability by the applications. The AEC Display Program provides an X-Windows based geographic display of the tactical situation. AEC operators may sign-on to any of the implemented sub-modes at any given display, allowing them to interact with the AEC system. The sub-mode positions currently implemented are Anti-Air Warfare Coordinator (AAWC); Tactical Actions Officer (TAO); Tactical Information Coordinator (TIC); and Computer Program Information Supervisor (CPIS). Once signed on, an operator may perform any action allowed for that sub-mode including listing, activating and displaying doctrine regions; initiating and monitoring engagements; reviewing alerts; and monitoring program status. The display window contains map display, alert and tutorial areas, a VAB/FAB panel, a close control CRO, a tactical CRO area, and a menu for map actions such as re-centering and re-scaling. Most of the functionality provided by the display program can be tailored via data files.

3.1.2.3 Display State Data Server

This server is a repository of information about the status of all AEC displays. It retains data on the location of each sub-mode signed on and maintains a queue of unreviewed alerts associated with the various sub-modes. It tracks display parameters, such as the most recently pressed VAB/FAB, currently displayed CRO, and currently hooked track at each display. Services are provided for the Display State Server clients to retrieve this information, allowing tactical and support programs to update CROs or act upon VAB actions as appropriate.

3.1.2.4 Manual Engage Control

The Manual Engage Control program is the interface between the operator and the engagement processing programs. It supports manual engagement of tracks, the subsequent sending of engagement requests to the Engagement Server; and the processing of alerts. Manual Engage Control is an Engagement Server client and receives updated target engagement status information from that server. It is responsible for updating operator CRO displays with current engagement status data as engagements progress.

3.1.2.5 Plan Server

a. The Plan Server maintains a data file of active and inactive doctrine statements and provides those statements to client programs on start-up and by request. For Demo 98 the Plan Server was extensively modified to incorporate Common Object Request Broker Architecture (CORBA) technology. The main objective was to begin the process of assessing the feasibility of using COTS-based DOC technologies for Navy mission-critical systems. The Doctrine/Plan Server and its client processes were identified as an area where the use of the CORBA technology could be investigated and demonstrated without adversely affecting any of the real-time requirements of the overall system. Modifications were made to the Doctrine/Plan Server program and its client programs (Auto-SM, Semi-Auto and Auto-Special) to use the CORBA technology. Previously, the interface between the Plan Server and its clients used Isis group communications middleware.

b. CORBA is primarily a distributed system object-oriented integration technology, and as such, the Doctrine/Plan Server and clients are thought of as objects in the distributed system. Client objects send requests and server objects respond to requests. In CORBA the Interface Definition Language (IDL) is used to specify the interface between objects, that is, what services the server provides to the clients. In this demonstration, the Plan Server and its clients were legacy applications, so that the interface and communication style were already somewhat decided. The interface consisted of the weapon doctrine data that the clients request from the Doctrine/Plan Server. The communication style provided by Isis was asynchronous messaging. At the time that this experiment was done, the latest standard was CORBA 2.0. This version did not include specifications for asynchronous messaging. Therefore, in this demonstration, the communication style used was synchronous request/reply where the clients request the weapon doctrine data and then block while awaiting the reply from the Doctrine/Plan Server.

c. The Plan Server was modified to use CORBA to receive requests (via the Object Request Broker (ORB)) from the clients for the weapon doctrine database. In addition, the ability for non-CORBA clients to request the weapon doctrine data using Isis was left intact. The clients were modified to use CORBA to make their requests (via the ORB) to the Plan Server by invoking the IDL interface operation getWeaponDoctrine on the CORBA Plan Server object. The ORB is responsible for ensuring that this invocation results in a call to the corresponding remote CORBA Plan Server object method, and returns the results to the client.

3.1.2.6 Semi-Auto

Semi-Auto receives tactical weapon doctrine from the Plan Server, then evaluates tracks against the currently active Semi-Auto doctrine regions. If a track's evaluated intercept point qualifies, a Recommend Engage alert is queued for the AAWC sub-mode. Semi-Auto is an Engagement Server client and as such, receives updated target engagement status information from that source. Semi-Auto is a peer client replicated for fault tolerance and load sharing[10].

3.1.2.7 Auto-SM

Auto-SM receives tactical weapon doctrine from the Plan Server and evaluates tracks against the currently active Auto-SM doctrine regions. It sends an engagement request to the Engagement Server upon track qualification. Auto-SM is an Engagement Server client and receives updated target engagement status information from that source. Auto-SM is a peer client replicated for fault tolerance and load sharing. [10]

3.1.2.8 Auto-Special

Auto-Special receives tactical weapon doctrine from the Plan Server and operates in two independent modes, SPY and AEC. The SPY mode evaluates tentative track and resolution messages reported by Track Correlation and Filtering (TCF) in response to the Auto-Special Parameters messages sent to Sensor Sim by the Plan Server. Auto-Special attempts to qualify the tracks supplied by TCF against the active Auto-Special statements. If a track qualifies, notification is forwarded to the Engagement Server. Auto-Special also has an AEC mode that periodically evaluates all tracks against the active Auto-Special doctrine statements and forwards an engagement request to the Engagement Server if a track qualifies. Auto-Special is an Engagement Server client and receives updated target engagement status information from that source. Auto-Special is a peer client replicated for fault tolerance and load sharing with respect to the AEC mode.[10] With respect to the SPY mode, Auto-Special uses an active replication model. Each Auto-Special replica acts independently on the tentative track and track resolution messages received from TCF, and forwards engagement requests to the Engagement Server as appropriate. The Engagement Server is responsible for filtering out duplicate requests. In this way, rapid reaction to significant threats is possible even upon failure of an Auto-Special replica.

---

[10] See HiPer-D Engineering Testbed Two (T2) Report, 30 Aug 96 for peer client architecture and implementation details.

3.1.2.9 Engagement Server

a. A major addition to the HiPer-D system in Demo 98 was the replication of the Engagement Server component. Prior to Demo 98 this component was not replicated and, therefore, represented a single point of failure in the HiPer-D engagement capability. This section describes this Engagement Server work and its affects and impacts on Demo 98. Table 3.1.2.9-1 briefly highlights the improvements made to the Engagement Server since the T3 Demo in August 1997.

Table 3.1.2.9-1 Engagement Server Improvements

T3 Demonstration (8/97)	Demo 98 (9/98)
1 Engagement Server; single point of failure	3+ Engagement Servers; primary/shadow execution and recovery model
No, or minimal, tactical state data in replicas; (RTDS Track sets)	Critical engagement state data of system-wide interest
Fault injection precision was approximate; Ctrl C, Kill, etc.	Precise fault injection capability that is data file driven; application tailored
No displays demonstrating consistency among replicated components	JEWEL display providing a window into replica state and action consistency

b. Execution Model: The purpose of the Engagement Server in HiPer-D is threefold. First, it validates engagement requests from clients and arbitrates any race conditions occurring due to multiple engagement requests on the same target. Second, it generates engagement orders to WCSSim for valid engagement requests. Third, it distributes engagement status updates to clients as tracks progress through their engagement sequence. During normal operation, doctrine clients (the Semi-Auto, Auto-SM, and Auto-Special applications) and tactical operators (the Manual Engage Control application) generate engagement requests on threatening tracks. The Engagement Server performs validation checks on these requests. Validated engagement requests result in engagement orders generated by the Engagement Server to the WCSSim. Finally, as the target cycles through the engagement sequence the engagement status for the target is updated and distributed by the Engagement Server. As previously mentioned, prior to Demo 98 the Engagement Server represented a single point of failure for the engagement capability of HiPer-D.

The Engagement Server design and execution relies on several features provided by the underlying group communications middleware. For Demo 98 this middleware was Isis, the group communications toolkit developed at Cornell University. The relevant features are listed below:

(1) FIFO message delivery between a transmitter and a receiver.

(2) Reliable and atomic delivery of message transmitted to a process group (i.e. all surviving members or no members receive the message).

(3) Group membership events ordered with respect to the message flow in the group.

c. FIFO ordering guarantees that a receiver will receive messages in the same order that a given transmitter sends them. Reliable and atomic delivery guarantees that either all members of a process group will receive a message or none of them will. This is an important feature to ensure that all group members have the same set of input messages. These two features together guarantee that all members of a group see messages in the same order from a given transmitter to that group. This provides an important guarantee of atomic ordering from a given transmitter. The ordered group membership events guarantee that membership changes to a group are seen by surviving group members at the same position in the message flow occurring for that group. For example, assume one member of a group receives a message followed by a subsequent membership change in that group. All surviving members of that group will see that identical ordering. In other words, it is guaranteed that all members will receive the message followed by the membership change. This is a critical feature to support fault tolerance in the presence of process failures. These features will be discussed again in Section 4.3 describing lessons learned in the Engagement Server work.

d. The Engagement Server design relies on these three group communications attributes to implement a semi-active primary/shadow execution and recovery model. The primary/shadow model requires that one replica be designated as primary. The primary replica is responsible for carrying out all processing and initiating any resultant responses. All other replicas are designated shadow replicas. The shadow replicas duplicate the computations of the primary but do not initiate any output except when the primary fails. The execution model and its dependence on group communications are discussed in the following paragraphs.

e. The first Engagement Server replica coming on-line assumes the role of the primary replica. All additional replicas come on-line in the role of shadow replicas. Demo 98 ran with three replicas, one primary and two shadow replicas. The design is not limited to this number, but there will always be only one primary replica. The primary replica has these unique responsibilities:

(1) Informs all shadow replicas that it has assumed the role of the primary replica.
(2) Informs all shadow replicas which message (i.e. input stimulus) to begin processing.
(3) Transmits messages to other components, as well as to shadow replicas, that result from processing an input stimulus. (These transmitted messages are called "resultants")
(4) Informs all shadow replicas when time-out conditions occur with respect to engagements in progress.

f. The shadow replicas have responsibilities that differ from the primary:

(1) Receives an indication of which replica has assumed the primary role.
(2) Receives and processes indications from the primary replica as to which input stimulus to begin processing.
(3) Verifies that the primary replica successfully transmits all required resultants.

(4) Receives and processes indications of time-out conditions that occur at the primary replica with respect to engagements in progress.

g. Table 3.1.2.9-2 summarizes the unique responsibilities of the primary and shadow replicas. These responsibilities should be considered in parallel and will be discussed to demonstrate the execution model. The first responsibility for both primary and shadow replicas allows all replicas to know which process has assumed the primary role. This is done at process initialization. The first replica detects that it is the only Engagement Server on-line and assumes the role of the primary replica. Additional Engagement Servers detect that there are other replicas on-line and wait to receive the indicator message from the primary replica. The primary replica detects these new replicas and then transmits this indicator to each of them. This allows the shadows replicas to properly identify the primary replica. This indicator is also transmitted after a failure of the primary replica but this will be discussed in regard to the third responsibility of primary and shadow replicas.

Table 3.1.2.9-2 Replica Responsibilities

Responsibility	Primary Replica	Shadow Replica
1. Assumption of Primary Role	Sends indicator to all shadow replicas.	Receives and processes indicator from primary replica.
2. Stimulus Eligible for Processing	Sends indicator to all shadow replicas when it begins processing a stimulus.	Receives and processes indicator from primary replica. Initiates processing of designated stimulus.
3. Transmission of Resultant	Transmits resultant to appropriate destination components and shadow replicas.	Receives and verifies transmission of resultant by primary replica.
4. Time-out Notification	Sends time-out notification to shadow replicas.	Receives and processes time-out notification from primary replica.

h. The second responsibility forces all replicas to process messages in the same order. Just prior to processing a message the primary transmits this information to the shadow replicas. This informs the shadow replicas which message to begin processing. The shadow replicas, in turn, are awaiting such an indicator to ensure that they process the identical message as the primary replica. The first and second Isis features, previously discussed, guarantee that each shadow replica will receive the specified message if they have not already. Once they receive this indicator they can proceed to process this message at the best possible speed. This technique of ordered message processing is an important step in ensuring that the replicas are staying in synch with respect to the engagement state data.

i. The third responsibility for each ensures that every resultant from processing an input stimulus does indeed get successfully transmitted. It is the responsibility of the primary replica to actually transmit the resultants. When the primary replica processes a message it simply

transmits a resultant at each point where that is necessary and then continues processing. The shadow replicas perform somewhat differently. When shadow replicas reach the identical point in processing a message they store this resultant in a pending-outbound state and then continue processing. The shadow replicas will receive each resultant transmitted by the primary replica, because of the ordering and atomitity guarantees provided by the group communications middleware first and second features of Isis. As they do so they are able to match these resultants against ones they have stored in a pending-outbound state queue. Matched resultants are subsequently removed. By this technique the shadow replicas can verify successful transmission of the resultants.

j. Should the primary replica fail before transmitting all resultants, the Engagement Server relies on the group communications guarantee of ordered membership change events with respect to the message flow in a group. With this guarantee, all surviving shadow replicas will know exactly which resultants the primary replica transmitted prior to failure. One shadow will assume the primary replica role. It will transmit to the surviving shadow replicas an indicator that it is now the primary replica. It will then transmit any resultants that remain to be sent and continue on in the role of the primary replica. The shadow replicas will await this primary replica indication. When they receive it they will then continue on in their role as shadow replicas. But, they will now perform their activities, including the matching of resultants, by looking for indicators from the new primary replica.

k. Finally, the fourth responsibility allows all replicas to be in agreement with respect to time-out conditions. When an engagement order resultant is transmitted to WCSSim, the Engagement Servers await a response from WCSSim. It is possible for this expected response to time-out. Upon transmission of an engagement order the primary replica sets a timer. If this timer expires before it receives the engagement response from WCSSim then the engagement has timed-out. The primary replica will transmit a message to the shadow replicas indicating whether the engagement response was received in time or it timed-out. In this manner, the shadow replicas can determine whether they should continue the normal processing for this engagement or proceed into the time-out processing. Either way, they will mimic the primary replica via the indicator it transmits to them.

l. Although the shadow replicas utilize the timeout notification from the primary replicas, they each also set their own timer when an engagement order is queued in a pending-outbound state. As long as the shadow replicas receive the time-out indicator message from the primary replica they can ignore and clear their own timers. These shadow timers are only needed when the primary replica fails. In this case, the shadow that becomes the new primary replica will then use it's own timers to determine all time-outs and send appropriate indicators to all shadow replicas.

m. The responsibilities defined above allow the creation of a replicated engagement service. This service is able to withstand process failures to any of the replicas, including the primary. Even in the case of a primary replica failure the engagement service capability will continue uninterrupted. This replicated engagement service is transparent to clients. Clients do not need to know how many Engagement Server replicas are cooperatively executing to provide this fault-tolerant service.

3.1.3 Track Data Services Components

The Track Data Server component provides a system wide unique track number through the Track Number Server, distributes correlated and filtered track data to all clients requesting the information via the Radar Track Data Server (RTDS), and provides the sensor feedback on frequency of update for priority tracks during track data overload conditions using the Sensor Rate Server (SRS). All track data service components are summarized in Table 3.1.3-1 and described in the following subparagraphs.

Table 3.1.3-1 Track Data Services Components

COMPONENT	*FUNCTION*
Radar Track Data Server (RTDS)	• Open client-server approach to radar track distribution • Provides client specific track groupings, report rate selection, and latency specification • Distributed implementation for scalability and fault tolerance
Track Number Server (TNS)	• Provides single number source for all components
Sensor Rate Server (SRS)	• Provides desired report rate for each Track to sensor simulator • Graceful degradation during track overload conditions

3.1.3.1 Radar Track Data Server (RTDS)

a. The RTDS was designed to serve as a flexible central distribution point for the radar track data produced by HiPer-D Correlation and Tracking (HCT). Its primary objective is to shield the radar track data recipients (e.g., displays, engagement review functions, etc.) from the excessive volume of sensor data produced by the TCF function, yet still provide the data necessary to ensure timely system operation.

b. RTDS provides an extremely flexible interface to its clients, allowing arbitrary groupings of tracks for specific track report rates and delivery latency needs. For example, when changing the range or center of a display, the display can request received tracks be displayed at a one-second rate, while other tracks (not even on the screen) can be updated at a five-second rate to reduce overall load on the display program. In the case of Auto-Special doctrine processing, where low track latencies are crucial, the RTDS client could request to receive track reports at a low latency setting for all those tracks in the vicinity of qualification. This results in very high responsiveness for critical needs, while reducing overall system processing and communications. It also provides more traditional query/response services in which the clients receive data only when requested. This is put to effective use by the sensor simulator which must, on notification of an engagement by WCS, retrieve kinematic data on the targeted object to compute properly an intercept trajectory and generate the contacts for the simulated ownship missile.

c. The RTDS is implemented as a replicated server. The replication provides fault tolerance, as well as distributes the communications load among all the server replicas. In the event that one of the replicas crashes, the remaining servers assume its load with no effect on the clients.

3.1.3.2 CORBA Track Number Server (CTNS)

CTNS is the Track Number Server (TNS) software adapted to utilize CORBA-based communication. The goal was to take an existing system having a sizable amount of

communication with other processes, both local and remote, and utilize CORBA[11] as the interface methodology. A brief description of the CTNS architecture and a performance summary are presented.

3.1.3.2.1 CTNS Overview

a. The CTNS provides a central location for managing unique track identifiers. The underlying concept is to have a single source (CTNS) maintain the track numbers, assuring that no number is ever used to identify multiple objects, that numbers do not get "consumed" never to be reused, and that numbers are reused on a least recently used basis. The following terms are useful when discussing CTNS functionality:

(1) Process Group Communications Layer – provides communications group membership services and supports communication between distributed groups.
(2) Process Group – A communications group implemented via the Process Group Communications Layer.
(3) CTNS Group – The CTNS Broker, CORBA_TNS, and CORBA_TNS_Bridge processes that work together to perform the TNS functions.
(4) TNSS Client – A TNS Service (TNSS) client that uses CTNS services.
(5) TNSS Group – A process group that contains all the members of the CTNS group and a particular TNSS client. There is one TNSS group for each TNSS Client.

b. For a TNSS client to acquire track numbers, it must join a TNSS group. This is accomplished through a request to the CTNS. The TNSS client is considered by CTNS to be a single entity for track number messages. Figure 3.1.3.2.1-1 shows two clients that have established communications with the CTNS through separate TNSS groups. The Process Group Communications Layer provides the functionality for joining a TNSS group and sending messages within a TNSS group.

[11] CORBA – Common Object Request Broker Architecture is an infrastructure that provides the services necessary for distributed objects to communicate without the user having to deal with network programming details. The CORBA standards are maintained by the Object Management Group (OMG).

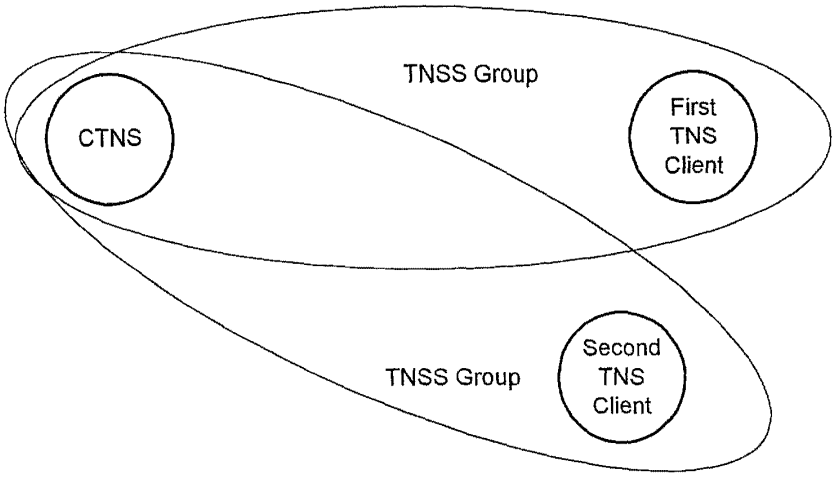


Figure 3.1.3.2.1-1 CTNS and TNS Client Relationship

3.1.3.2.2 Overall Architecture

a. CTNS was implemented using Iona Technologies Orbix CORBA environment. CTNS comprises the following modules:

(1) corba_tns process – This is the main program in the CTNS. It is a C++ program which is started by system control, and is responsible for managing the Track Number database. When a TNSS client signs on to the TNS process group, corba_tns creates an object for the TNSS client. This object is used for managing the client's track number data.

(2) ctns_broker process – An ANSI C program that manages TNSS client messages with the CTNS. This program is started by system control.

(3) corba_tns_bridge subprocess – A C++ program that provides an interface between corba_tns and the ctns_broker. This program is launched by ctns_broker. It acts as a bridge between the process group communications based clients (coded in ANSI C) and the CORBA-based server (coded in C++).

(4) Object Request Broker (ORB) - The middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located or any other system aspects that are not part of an object's interface. The ORB is a standard component of CORBA.

(5) orbixd daemon – This is an Orbix-furnished process that initiates communication between corba_tns_bridge and corba_tns. It provides an initial object reference[12] to

[12] Object reference – The CORBA Object Request Broker (ORB) uses object references to identify and locate objects so that it can direct requests to them. As long as the referenced object exists, the ORB allows the holder of an object reference (client) to request services from it. It appears as if the client is making a local procedure call, when in fact the object can be located on a remote processor.

corba_tns_bridge so that corba_tns_bridge can make method invocations on corba_tns. Orbixd is launched by system control. Orbix requires that orbixd be collocated with each instance of a CORBA server. While this process is specific to Orbix, other CORBA packages furnish this capability in one form or another.

b. The CTNS architecture is illustrated in Figure 3.1.3.2.2-1:

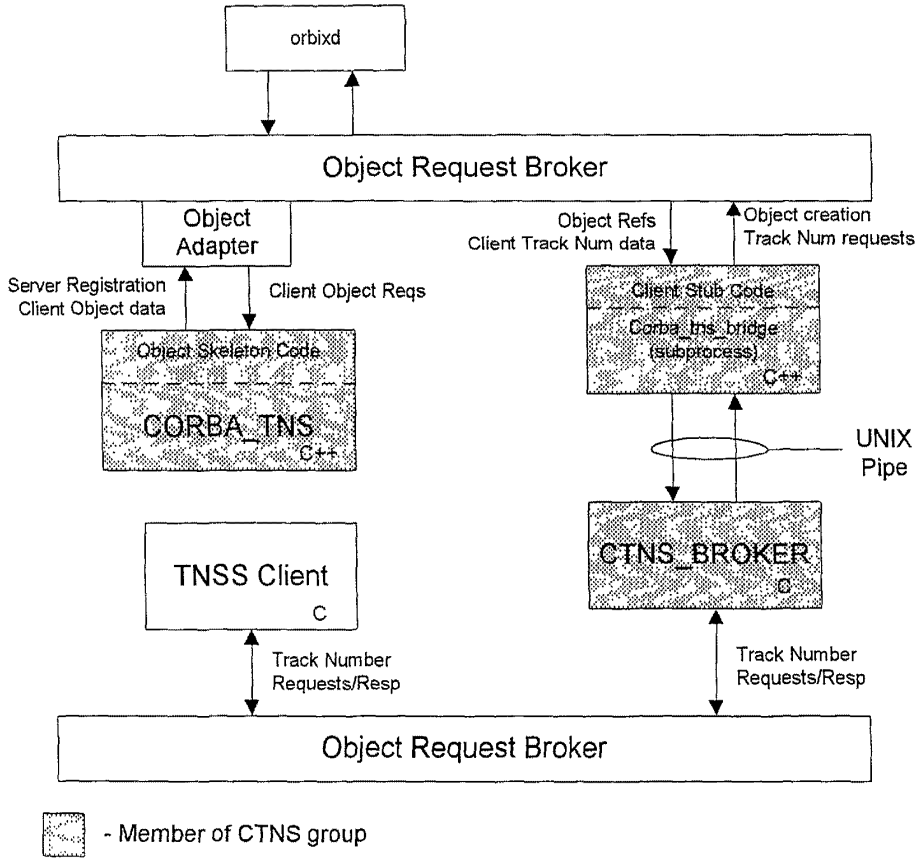


Figure 3.1.3.2.2-1 CTNS Architecture

3.1.3.2.3 CTNS / TNSS Client Communications

a. Communication between the CTNS group and a TNSS client is via the ctns_broker module. This is a process group communications based ANSI C program that is responsible for the following:

(1) Initiating TNSS group sign-on via the process group communications layer
(2) Accessing the Track Number Server database (corba_tns module) via the corba_tns_bridge subprocess

(3) Providing a destination for TNSS client track number requests made via the process group communications layer

b. Process group communications layer based applications are multithreaded in the sense that the process group communications layer will issue multiple up-calls in parallel in response to messages in multiple groups. After initialization, ctns_broker threads are created for starting a TNS reference group and processing client messages for TNSS group sign-on. ctns_broker threads are created for each TNSS client that joins a TNSS Group. The threads are responsible for processing requests made by the TNSS client as well as monitoring the group status. All communication between the ctns_broker threads and a TNSS client is via the process group communications layer.

3.1.3.2.4 CTNS Group Communication

a. The interface to the CTNS server is defined with CORBA's Interface Definition Language (IDL).[13] Based on an IDL file, the Orbix IDL compiler generates modules for the CORBA server (Object Skeleton code) and client (Client Stub code) to interface with the ORB. The Object Skeleton code is the framework for the methods to access the objects specified in the interface. The developer provides the code for these methods in the Object Skeleton. The Client Stub code is complete when generated by the IDL compiler. This code provides the mechanisms needed by the CORBA client to access the server's objects, either remotely or locally, via the ORB. The ORB is part of the underlying CORBA infrastructure that handles the communication between the CORBA server and client. IIOP[14] is the protocol used for the communication between the CORBA server and client. The CORBA server and client are not required to be on the same processor nodes; communication between the two processes will function regardless of their locations due to the ORB.

b. The CORBA server responsible for managing the track number database is corba_tns. It creates objects for each TNSS client that joins the TNSS group via ctns_broker. The corba_tns_bridge subprocess (CORBA client) invokes these objects in response to track number data requests from ctns_broker. corba_tns and corba_tns_bridge are not required to be on the same processor nodes; communication between the two processes will function regardless of their locations.

c. In order for communication to be established between the distributed members of the CTNS group, the following must occur. When corba_tns starts up, it notifies orbixd via the ORB, that it is ready to receive data requests. corba_tns_bridge requests orbixd to provide an object reference for corba_tns. The CORBA client / server communication is now established.

---

[13] IDL – A language similar to C++, used to specify the objects and object methods to be utilized in accomplishing certain tasks in a distributed environment. An IDL specification is independent of the language used to implement either the client or the server.

[14] This is the Internet Inter-ORB Protocol, which specifies interoperability between different ORB implementations. IIOP is the standard CORBA communication protocol. Under IIOP, an object reference is provided in the CORBA Interoperable Object Reference (IOR) format. The main components of an IOR are the host name on which an object's server resides, the port number on which the server is listening for IIOP traffic, and an object key, which uniquely identifies an object within a server.

As long as corba_tns has an object established, and corba_tns_bridge holds a reference to that object, corba_tns can process data requests from ctns_broker.

d. Upon receipt of a TNSS client data request, a ctns_broker thread sends the information over a FIFO pipe. The corba_tns_bridge subprocess is blocked on a read of the pipe. The corba_tns_bridge invokes a method on the specified TNSS client object corresponding to the data request it receives over the pipe. The corba_tns method is invoked and data is exchanged based on communication between the Client Stub code, ORB, Object Adapter, and Object Skeleton code. This communication is transparent to the CORBA server and client. corba_tns_bridge passes the data it receives back through the pipe to ctns_broker.

e. Initially, it was intended that the ctns_broker program be a CORBA-based C++ application that would communicate with corba_tns via the ORB. It would still utilize the process group communications layer to communicate with TNSS clients. This was not possible due to incompatibilities with C programs calling C++ programs (the process group communications layer is written in C). To get around this obstacle, ctns_broker remained a Spread-based C program. It forks a CORBA-based C++ subprocess (corba_tns_bridge), with which it can communicate over a Unix FIFO pipe. This allows a standard C application to communicate with the CORBA-based C++ application (corba_tns), although it adds an extra level of complexity and inefficiency.

3.1.3.2.5 Startup Processing

a. At startup, the corba_tns program registers with orbixd to establish its location and identify an initial object reference[15] to the ORB. It then awaits a data request. After corba_tns_bridge starts up, it requests an object reference for corba_tns, which is provided by orbixd. After ctns_broker accepts sign on to the group of a TNSS client, it notifies corba_tns_bridge of the new client. corba_tns_bridge then uses its initial object reference to request that an object be created for the new TNSS client. corba_tns provides a client object reference to corba_tns_bridge for the new TNSS client. Any data requests related to this TNSS client are to be made using the client's object reference. The passing of data and any method invocations between corba_tns and corba_tns_bridge are handled by the ORB and are transparent to the programs.

b. After corba_tns has started, the ctns_broker program forks a subprocess that starts the corba_tns_bridge program. Ctns_broker establishes a pipe between itself and corba_tns_bridge to provide for communication between the two programs. It then calls the gms_join routine which initializes TNS process group communications using the HiPer-D DCS/GMS interface layers. It then registers routines with the tns_if module to accept TNS domain sign on messages and TNS domain messages, and signs on to the TNSS Group. Control is then passed to process group communications, which calls the above-mentioned routines when messages arrive from a TNSS client. Ctns_broker places each message it receives in a FIFO pipe to be read by corba_tns_bridge.

[15] The initial object created by corba_tns is a factory object. When corba_tns_bridge binds with corba_tns, it receives an object reference for this factory object. The factory object provides methods for creating and destroying client objects. When a new client object is to be created, corba_tns_bridge invokes a method on this factory object.

c. When the corba_tns_bridge program is started, it tries to bind with the corba_tns program via the ORB. This provides corba_tns_bridge with the initial object reference that it needs to communicate with corba_tns. Once successful, corba_tns_bridge blocks on the CTNS message buffer waiting for a message from ctns_broker.

3.1.3.2.6 Performance

a. The CTNS interface was implemented with the capability to provide new track numbers to a client and to release a track number held by a client.

b. The performance of the CTNS interface was evaluated with the CTNS processes located on the same node, as well as distributed on different nodes. Distributing CTNS requires that corba_tns and orbixd be located on the same node. The CTNS was tested with multiple TNSS clients, and the maximum track number capacity (10,000) was verified for at least one client. To determine if the CTNS interface was functioning properly, the following characteristics were observed as tracks were entered in the system:

(1) The log files for ctns_broker, corba_tns_bridge, and corba_tns were evaluated to verify that:

(a) ctns_broker signs on to a group properly and a client object is created for each TNSS client that signs on to the group

(b) Messages from a TNSS client are formatted properly within the CTNS group, the client object is updated correctly for the given message, and results of the message are returned to the client properly.

(2) The display (Ximp) was observed to verify that the number of tracks in the system corresponded to what CTNS had allocated.

(3) Modelc log file showed a normal distribution of track reports while CTNS was allocating track numbers to the system.

3.1.3.3 Sensor Rate Server (SRS)

The SRS provides an opportunity to use the available sensor bandwidth in an intelligent way. For example, there may be limits in sensor or processing capabilities due to hardware and/or software configurations. These limits may be temporary or permanent. The SRS adapts the requested sensor data load by collecting information on sensor capabilities, desired track report rates, and track importance. It then maps requested rates and priorities onto sensor capabilities and issues commands to the sensor(s), specifying the rate at which each track is to be reported.

3.2 Land Attack and C^4I Subsystem Functional Description

A land attack capability provided by the ATWCS Real Time Launch Control function is a new component added to the test bed for Demo 98. The path for OTH tracks to be entered into the testbed is provided through the JMCIS and AACT components. The ABMX component and

the data brokers providing the interfaces to these systems are part of this subsystem. These components are highlighted in Figure 3.2-1 and described in the following subparagraphs.

3.2.1 Advanced Tomahawk Weapons Control System (ATWCS)

a. The ATWCS has a number of responsibilities. First, it is responsible for over water route planning for Block II and Block III Tomahawk Land Attack Missile (TLAM) strikes against predefined land targets. Secondly, ATWCS aligns the missiles inertial navigator and downloads flight software and mission data, enabling the missile to navigate to the target. It responds to Vertical Launching System (VLS) and missile fault conditions, maintains weapon safety and controls missile launch sequence. In addition, it responds to VLS and missile fault conditions, maintains weapon safety, and controls TLAM initialization and launch. Overland routes and terminal attacks are planned external to ATWCS by Tomahawk mission planning systems either ashore, at Cruise Missile Support Activities (CMSA), or on command ships at Afloat Planning System (APS) Detachments. These preplanned missions are stored in a library maintained by ATWCS. The library may be updated via either magnetic tape, or satellite communications from CMSA or APS sites.

b. ATWCS is composed of several major subsystems, or Computer Software Configuration Items (CSCI). The over water planning function is performed by the Engagement Planning and Control (EPC) CSCI, supported by the Command, Control, Communication and Intelligence (C^3I) CSCI which provides the near real-time surface track tactical picture. The missile initialization and launch function is implemented in the Launch Control (LC) CSCI, supported by the Mission Data Process (MDP) CSCI which provides the preplanned overland route and target data from the mission library. A System Management (SM) CSCI has overall control of the ATWCS system and provides general-purpose services such as mode (tactical, training or test) control and operator alert processing.

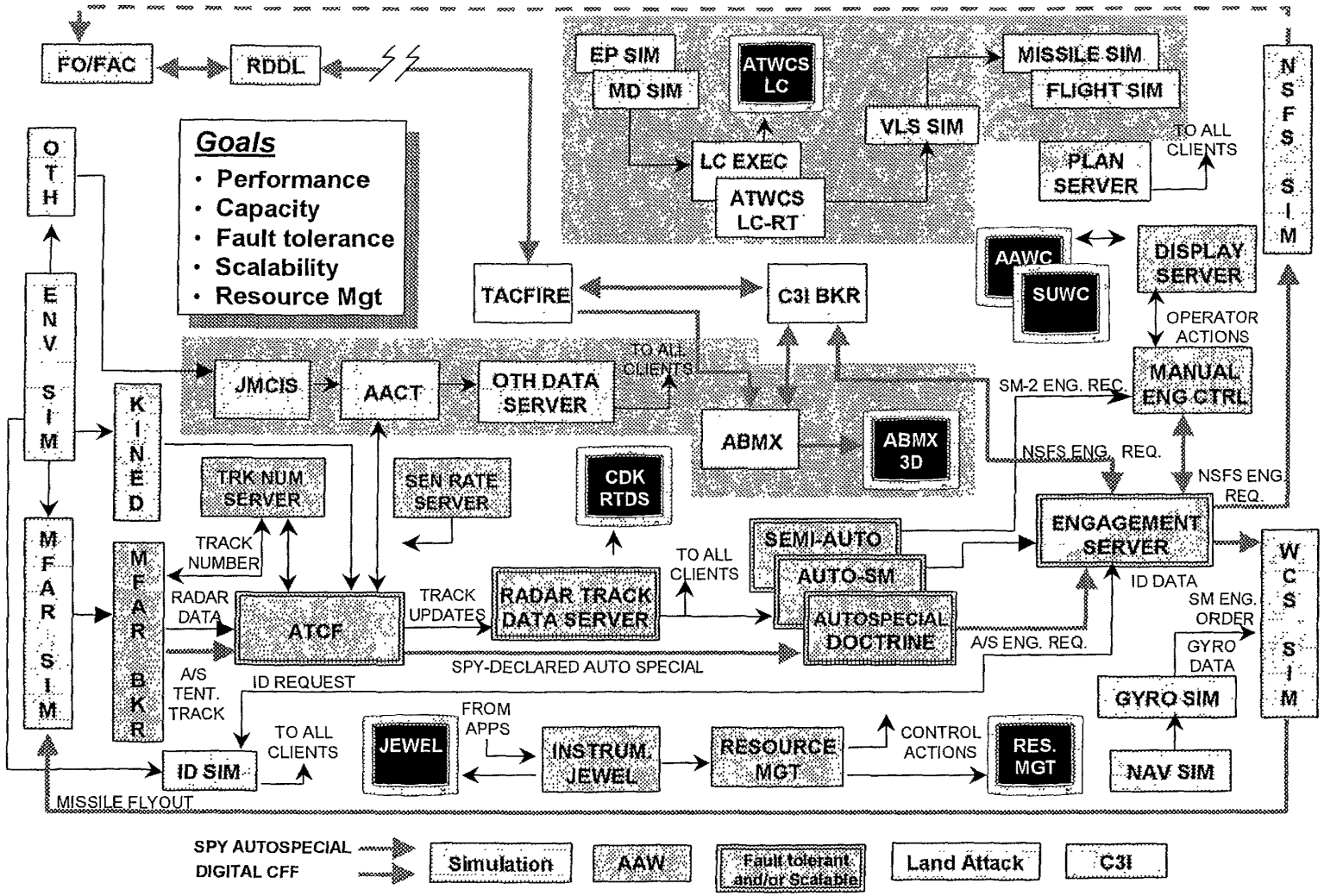


Figure 3.2-1 Demo 98 Block Diagram with ABMX and Data Brokers Highlighted

c. In support of the 98 Demo objectives, the following architectural concepts were investigated as part of the ATWCS element of the demonstration:

(1) Open systems design

(2) Distributed architecture

(3) Hardware independence (a.k.a., software portability)

(4) Shared resource management

(5) Support of real-time processing

(6) Fault tolerance

d. The LC CSCI was selected as the prototype effort for Demo 98. This decision was based on the following facts. The LC:

(1) Has an architecture that is conducive to supporting the demonstration objectives.

(2) Has well-defined real-time and non-real-time requirements.

(3) Was developed using multiple programming languages and operating systems.

e. Figure 3.2.1-1 presents a high-level view of the architecture of the operational ATWCS, with the LC CSCI shown decomposed into its three major processes, the LC Human Computer Interface (HCI), the LC Executive (Exec), and the LC Real-Time (RT). LC HCI provides the graphical interface to the LC operator; LC Exec is the "brain" of the LC CSCI, where the state of each active Tomahawk engagement plan is maintained. The LC RT process handles the communication between the ship's Inertial Navigation System (INS) (the forward and aft WSN-5s) and the VLS Launch Control Units (LCU).

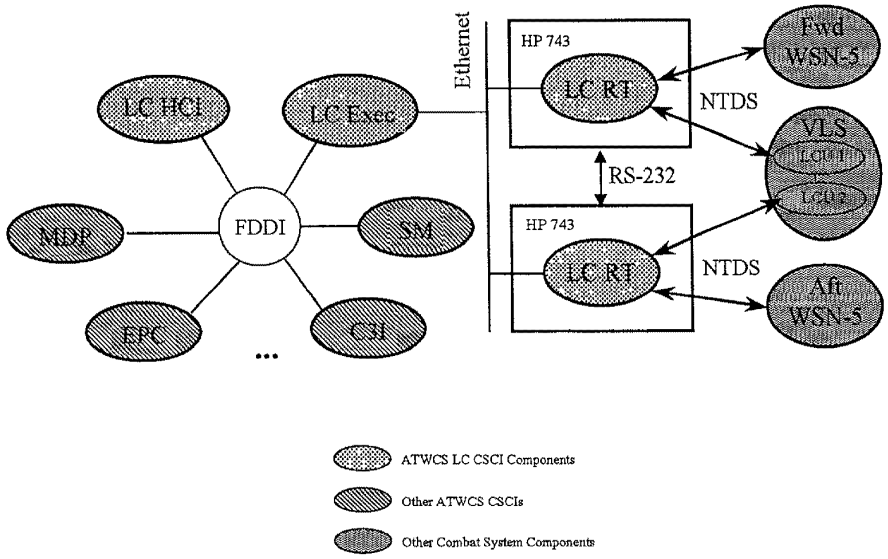


Figure 3.2.1-1. High Level Architecture of Operational ATWCS

f. With the exception of the LC RT process, ATWCS CSCIs are designed to run on TAC-3/4 (HP) processors with the HPUX 9.07 operating system. Communication between CSCIs (and LC processes) is implemented by Application Program Interface (API) software in a client/server architecture on a local area network. The LC RT process is designed to run on an HP 743 single board computer under the HPRT operating system. Communication between the LC RT processor and the ship's INS and VLS are point-to-point NTDS interfaces. In order to provide a degree of fault tolerance, the LC RT process is normally running simultaneously in two separate HP 743 processors. This configuration allows automatic recovery from HP 743 processor failure, loss of communication with the primary WSN-5, or loss of communication with either VLS LCU. However, multiple failures may require operator intervention to overcome, or may be unrecoverable. Figure 3.2.1-2 depicts the ATWCS/HiPer-D integration for Demo 98.

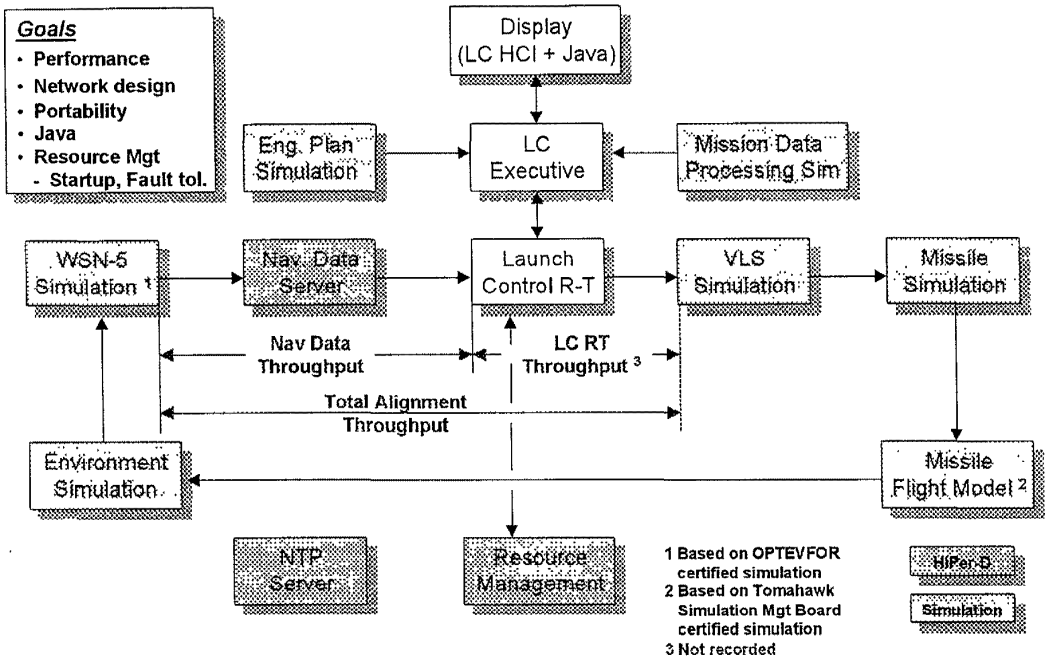


Figure 3.2.1-2 ATWCS Network Connectivity

g. The LC version 2.1 tactical software, one of the developmental builds for ATWCS, was selected as the baseline for the Demo 98 advanced computing prototype. The functionality of the other ATWCS CSCIs and combat system components was simulated. These simulations satisfy all of the LC interface requirements, but do not provide all of the operational functionality of their tactical counterparts.

h. Demonstrating portability was approached in two ways. First, the portable programming language, Java, was used for the LC HCI software because it allowed software to be developed on one platform and run on many platforms. The second approach was to identify the hardware dependencies in the LC RT software, and isolate those dependencies in such a way that only a recompilation would be needed in order to run the existing software on a Sun/Solaris platform.

i. Demo 98 also demonstrated the fault tolerance of LC RT in a distributed architecture, integration with the ship-wide computing Resource Manager, adoption of network connectivity among mission critical elements of the combat system, and simulation of the Tomahawk missile flight. In accordance with the vision for future combat systems, the point-to-point interfaces between ATWCS and simulations representing the ship's INS and VLS have, in the testbed architecture, been replaced with network connectivity.

j. Figure 3.2.1-3 illustrates the architecture of the ATWCS component of the Demo 98 testbed. None of the supporting infrastructure (NTP, JEWEL, RM, etc.) is shown in this diagram. As can be seen, a Nav Sim program running on an SGI processor, and a Nav Data

Server application running on a Sun simulated the ship's INS. The baseline LC Exec and LC HCI processes, along with non-real-time components of the other ATWCS CSCI simulations, were hosted on a TAC-4 under HPUX 10.2. In addition to a Sun for each of the prototype LC RT and LC HCI processes, another Sun was required to host the real-time components of the supporting simulations.

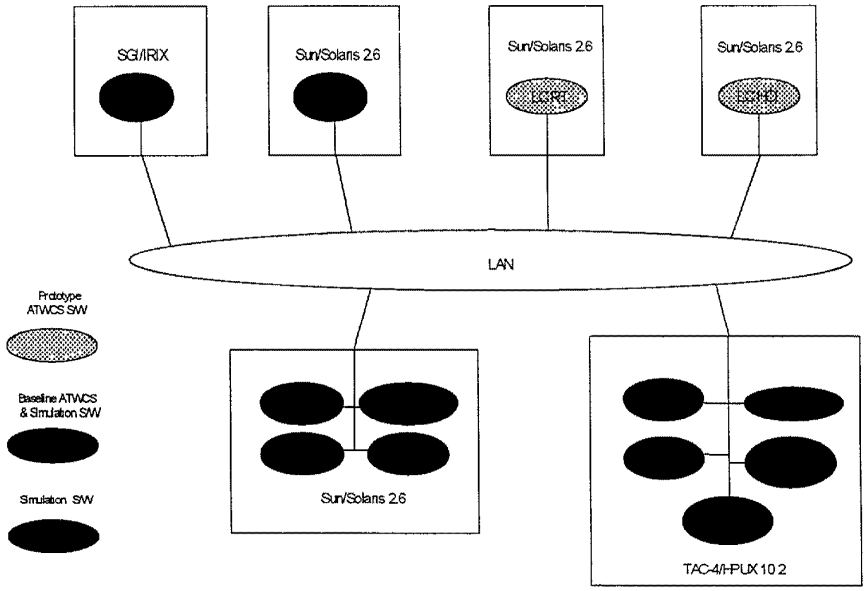


Figure 3.2.1-3 ATWCS Demonstration Testbed Architecture

3.2.2 Joint Maritime Command Information System (JMCIS)

a. The JMCIS was the source for non-organic track data (often referred to as over-the-horizon or OTH data). JMCIS is an automated C^4I system that interfaces to a variety of military communications and computer systems. JMCIS is designed to provide tactical situation assessment, data fusion and display capabilities to battle group and force commanders.

b. JMCIS is made up of a variety of networked computers. The central processor is known as JOTS1. JOTS1 receives data from other JMCIS components including those designed to handle communications interfaces. JOTS1 performs data correlation then distributes that data to all other JMCIS processors. All JMCIS processors communicate using the JMCIS TDBMS protocol over Ethernet LAN connections.

c. In Demo 98, there was a JOTS1 processor and a JOTS28 processor. JOTS28 is a general-purpose processor with no predefined role in the JMCIS network. In HiPer-D, JOTS28 hosted the Advanced Battle Management and eXecution (ABMX) software (formerly Advanced Power Projection Planning & Execution (APPEX)).

d. Demo 98 did not contain the JMCIS communication processors. An interface was developed to drive JMCIS that allowed tracks to be introduced into JMCIS using an Over-the-Horizon Targeting Gold (OTHGOLD) interface. OTHGOLD is an ASCII file format that contains track position information. OTHGOLD files are read by the JMCIS 'COM' program which passes the data to JOTS1. To simulate a moving target, a process called OTHSIM was developed. This process is illustrated in Figure 3.2.2-1 below.

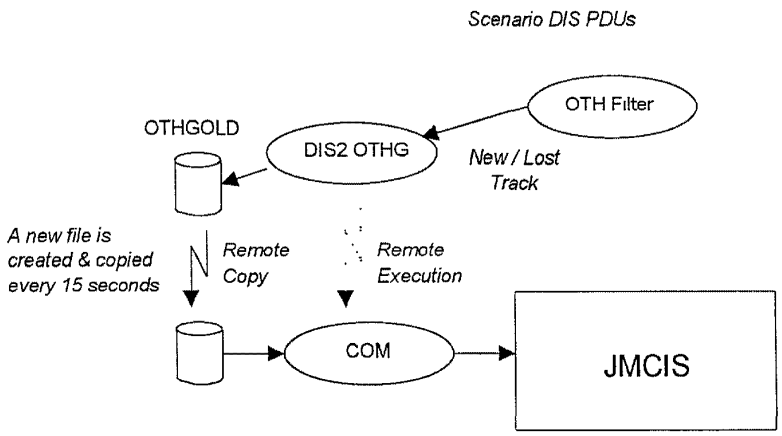


Figure 3.2.2-1 - JMCIS Stimulation

3.2.2.1 DIS to OTHGOLD Converter

a. The program dis2othg is a software filter that reads DIS PDUs (as specified in IEEE 1278.1 - 1995) as input from a Unix socket. It examines each PDU as it arrives to determine its type. If the PDU is an Entity State (ES) PDU, it proceeds to convert the appropriate PDU data fields to OTHGOLD message components. The program writes a file to disk containing the OTHGOLD messages (the "OTHGOLD file") triggered either by a timer or a certain number of message components having been prepared for writing. The DIS to OTHGOLD Converter architecture is illustrated in Figure 3.2.2.1-1.

b. Dis2othg has a companion Unix shell script, copy_oth_gold.csh, that copies the OTHGOLD file to a designated computer that then runs the JMCIS import_file program to get the OTHGOLD data into the JMCIS system.

c. The program and shell scripts coordinate their operations by setting and examining the file protection bits of the OTHGOLD file. This ensures that dis2othg will not overwrite the OTHGOLD file before it has been moved to the JMCIS computer or while the shell script is doing its copy operation. This scheme also prevents the shell script from attempting to copy an incomplete OTHGOLD file as dis2othg is writing it.

d. The usual arrangement is to have dis2othg and copy_oth_gold.csh running on the same computer. But this is not necessary. As long as the disk file can be seen by the computers that run the program and shell script, the desired data from DIS will be moved to JMCIS.

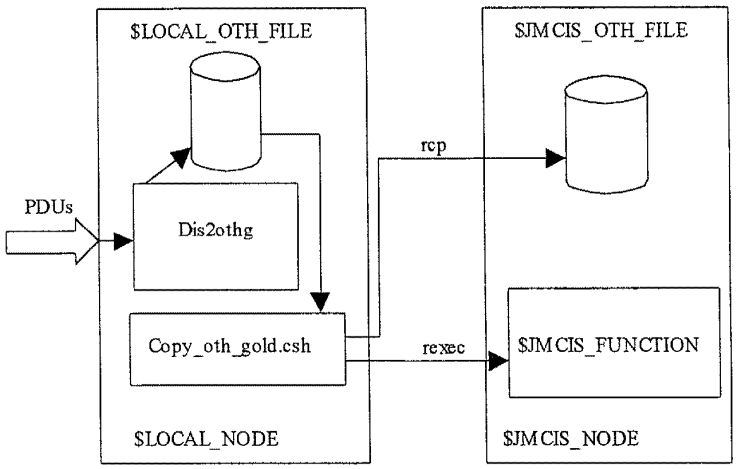


Figure 3.2.2.1-1 DIS to JMCIS Interface

3.2.3 Advanced Battle Management and Execution (ABMX) System

a. ABMX is a force-level mission planning, preview, and battle management system that allows carrier-based air wings to plan, visualize and assess strike plans prior to the launch of aircraft. ABMX began as an Advanced Combat Technology Demonstration (ACTD) and was successfully deployed for six months aboard USS Theodore Roosevelt (CVN-71).

b. ABMX offers a two dimensional (2D) display on the JOTS28, and a three-dimensional (3D) display resident on a Silicon Graphics workstation. The ABMX 2D is used for mission planning, fine-tuning, and battle management. The ABMX 3D display provides a 3D battle display. ABMX 3D in Demo 98 was used for visual deconfliction during the digital CFF sequence.

3.2.4 Data Brokers – Legacy System Interface

a. The key aspect of integrating legacy systems into the testbed is the ability to establish intercommunication among all the components. This was accomplished in Demo 98 with the AEGIS Air Correlator and Tracker (AACT) and the C^3I broker. AACT provides the interface between C4I, JOTS1, and the RTDS AAW components. The C^3I broker provides the interface between HiPer-D combat systems and ABMX and Naval Surface Fire Support Simulator (NSFS Sim). A track broker was developed to interface between AACT and the HiPer-D Track Correlator and Filter.

b. This section documents the JMCIS / AACT / HiPer-D interface architecture and software used in the HiPer-D 1998 Demonstration.

3.2.4.1 JMCIS/AACT/AAW Interface

Real-time track data and OTH track data flow through HiPer-D along two different paths. Real-time tracks flow from local radar sources, through the ATCF, to the RTDS, which in turn distributes them to its clients. JMCIS is the source for OTH tracks which are forwarded through AACT, where they are associated with real-time tracks. AACT uses the CDK track file to store both real-time and OTH tracks. The CDK track file record includes a field which is used to keep the OTH and real-time data separately identifiable. The interface to AACT was implemented by developing processes that accessed the CDK track file. This architecture is illustrated in Figure 3.2.4.1-1.

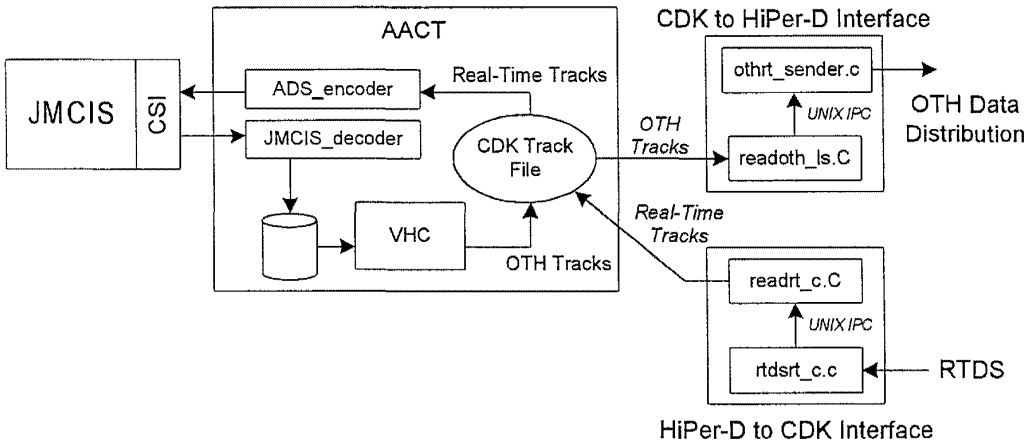


Figure 3.2.4.1-1 JMCIS / HiPer-D Interface

3.2.4.2 Real-Time Data AAW Track Path

a. AACT receives real-time tracks from the RTDS through the HiPer-D to CDK Interface shown in Figure 3.2.4.1-1 above. This interface stores the real-time tracks in the CDK track file. The Virtual Hypothesis Correlator (VHC) process in AACT finds associations between these real-time tracks and the OTH tracks that are received from JMCIS.

b. The RTDS interface is implemented using HiPer-D support libraries, which are written in ANSI C. The CDK track file is accessed using CDK libraries, which are written in C++. C++ routines cannot easily call C routines. This forces the HiPer-D to CDK interface to be composed of two modules that communicate using a Unix Interprocess Communications (IPC) mechanism known as a Unix pipe.

c. The module on the ANSI C side, *rtdsrt_c*, registers with the RTDS as a client and begins receiving real-time track information. It packages this information into the appropriate

track data structures (i.e., drop track, new track, or track update) and forwards them across the Unix pipe to the C++ side. The module on the C++ side, *readrt_c*, listens to the Unix pipe and decodes the messages that are received. The CDK track file is then updated with this information. Data conversions are made, as data is stored in the CDK track file to adapt for different units used by the RTDS and CDK. The *ads_encoder* process periodically scans the CDK track file and sends any real-time tracks that it finds to JMCIS.

3.2.4.3 OTH Track Data Path

a. The OTH track reports are forwarded by JMCIS to the CDK track file via the JMCIS Combat System Interface (CSI) segment interface developed by International Research Institute (INRI). The CSI is composed of an MDX process, which accesses decoder, and encoder processes, which are defined by the MDX user. AACT defines the *ADS_encoder* and the *JMCIS_decoder* routines to MDX. The *JMCIS_decoder* process decodes the messages received from JMCIS and writes the information to a file on disk. This disk file is then read by the VHC process, which performs the association and inserts the OTH tracks in the CDK track file. The CDK to HiPer-D Interface, as shown in Figure 3.2.4.1-1 above, reads these tracks from the CDK track file and distributes them to clients in HiPer-D

b. The *readoth_ls* component of the CDK to HiPer-D Interface is written in C++, and the *othrt_sender* component is written in ANSI C. These two processes use a Unix pipe to communicate. *Readoth_ls* periodically scans the CDK track file and picks out the OTH tracks by examining the *track_type* field for each track. This module keeps track of the existing OTH tracks to be able to determine if any of the tracks processed on the current scan are new, updated, or missing. Once this is determined, the appropriate new track, track update, or drop track is sent via the Unix pipe to *othrt_sender*. *Othrt_sender* listens on the Unix pipe, decodes messages that come across, and broadcasts them using HiPer-D's OTH data server communications group.

3.2.4.4 Aegis Air Correlator Tracker (AACT)

a. AACT interfaces between HiPer-D and JMCIS using the JMCIS CSI segment. When JMCIS transmits updates, AACT receives the update and saves it in a temporary data store. VHC uses an advanced correlation algorithm to associate JMCIS data with Real Time tracks from the Aegis combat system. These associations are placed in the AACT track file where they can be displayed by the Common Display Kernel (CDK) display. AACT periodically scans the AACT track file and sends real time track information to JMCIS.

b. The AACT track file contains both real time and JMCIS tracks. This track file became the focal point of the interface between C4I and AAW subsystems. This is illustrated in Figure 3.2.4.4-1 below.

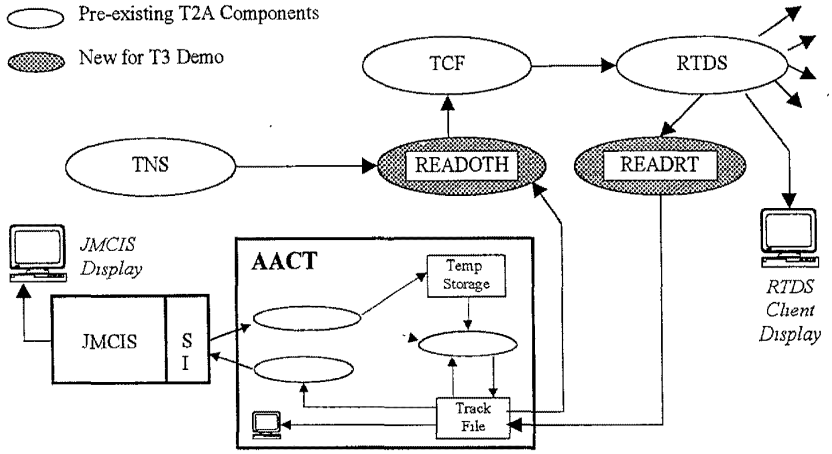


Figure 3.2.4.4-1. AACT to AAAW Interface

c. Every 20 seconds, the READOTH process scans through the AACT track file looking for JMCIS data that has been placed there by the VHC process. When it finds new track data, it acquires a HiPer-D track number from the HiPer-D TNS and assigns that number to the new track. It then forwards that track number to the HiPer-D TCF process. The TCF forwards the track to the RTDS which distributes it to other HiPer-D components. READOTH maintains a list of JMCIS tracks that it has passed on to the TCF. If it fails to see a track in the AACT track file during a scan, it assumes that the track has been dropped and issues a drop track message to the TCF.

d. The READRT process is a client of the RTDS and receives all tracks issued by the RTDS. READRT takes these tracks and places them in the AACT track file. This provides a path for passing real time data back into JMCIS.

e. During Demo 98, all inputs into both JMCIS and HiPer-D were scripted using HiPer-D's simulation capability. These scripts were deliberately designed so that JMCIS would not report tracks that would also be seen by the combat system. No correlation of JMCIS tracks and real time tracks was performed. (This was done because the architectural issues associated with introducing non-real time tracks into a real time combat system are very complex, and could not be resolved in the limited time available.) The architectural issues associated with properly integrating the combined data are left for future efforts.

3.3 Simulation and Support Components

One of the technical objectives for Demo 98 was to add a physics based DIS compatible wraparound simulation capability. This capability would allow the testbed to run more operationally oriented scenarios and would provide the ability to interconnect with other geographically dispersed organizations for future testing and demonstrations. The Environment Simulation components and other simulation components presented in this section meet that objective. The support components include time synchronization, near real time instrumentation, and system startup. These components are highlighted in Figure 3.3-1. The support components are summarized in Table 3.3-1.

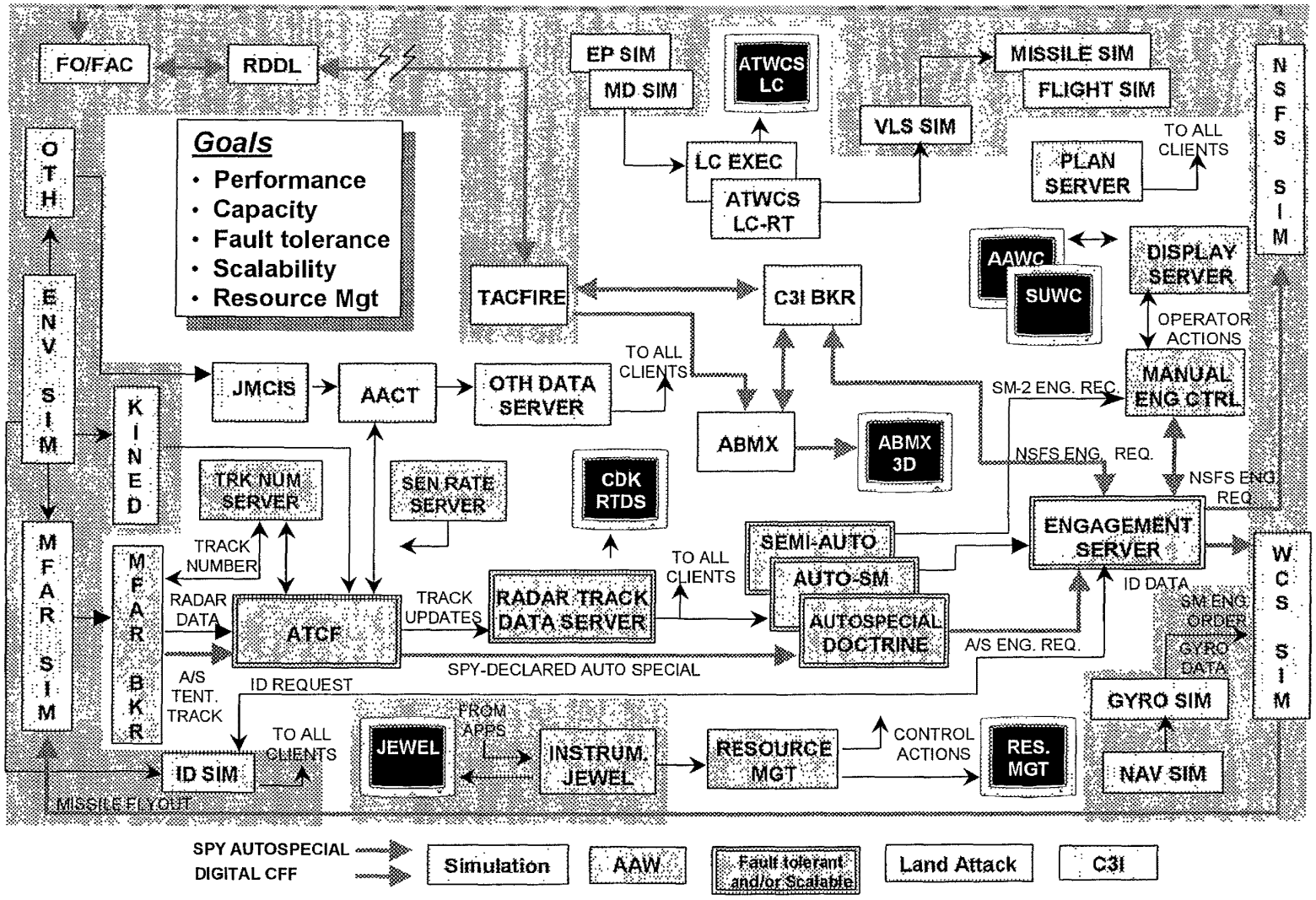


Figure 3.3-1 Demo 98 Block Diagram with Simulation and Support Components Highlighted

Table 3.3-1 Support Components

COMPONENT	*FUNCTION*
Simulation Control (SIMCON)	• Scenario Input Driver
Sensor Simulator	• Dynamic Track Generation Capabilities • Missile Track Generation for Intercept Simulation • Distributed Implementation • Provides Ownship Speed and Course at Selectable Rate
KINED	• Generates Periodic Reports on Tracks • Distribution Implementation
WCS Simulator	• Minimal Simulation of Aegis WCS • Initiates Missile Fly-out Upon Engagement • Probabilistic Kill / No Kill Assessment Decision • Client of RTDS
IDU Simulator	• Provides ID data to AEC upon Quick ID request or manual change of ID by IDU Sim operator • Client of RTDS
System Control	• Controls start-up of Track Data Services components • Monitors component health • Automatically restarts components on failure • Provides "plan language" for specifying component interdependencies and start-up order • Provides single point operator interface

3.3.1 Environmental Simulations (EnvSims)

a. Environmental Simulations (EnvSims) provide a wrap-around simulation capability with which to drive the Advanced Computing testbed. The EnvSims in Demo 98 differed from those used in previous demos in that, in Demo 98, physics based models were used. Secondly, the simulations complied with IEEE Standard 1278.1-1995 for Distributed Interactive Simulation (DIS) – Application Protocols. Complying with the standard will enable the EnvSims to interact with other DIS simulations off-site.

b. The EnvSims are broken down into the three categories listed below.

Entity Simulations

Platform Generator

Missile Generator

NavSim & Helm Control

SM-2 Flight Sim

Tomahawk Flight Sim

Sensors

Multi-function Array Radar (MFAR)

OTH Filter

DIS to Rainform Gold Converter

Displays

Truth Display

Vertical Profile Display

c. Figure 3.3.1-1 shows the EnvSim elements.

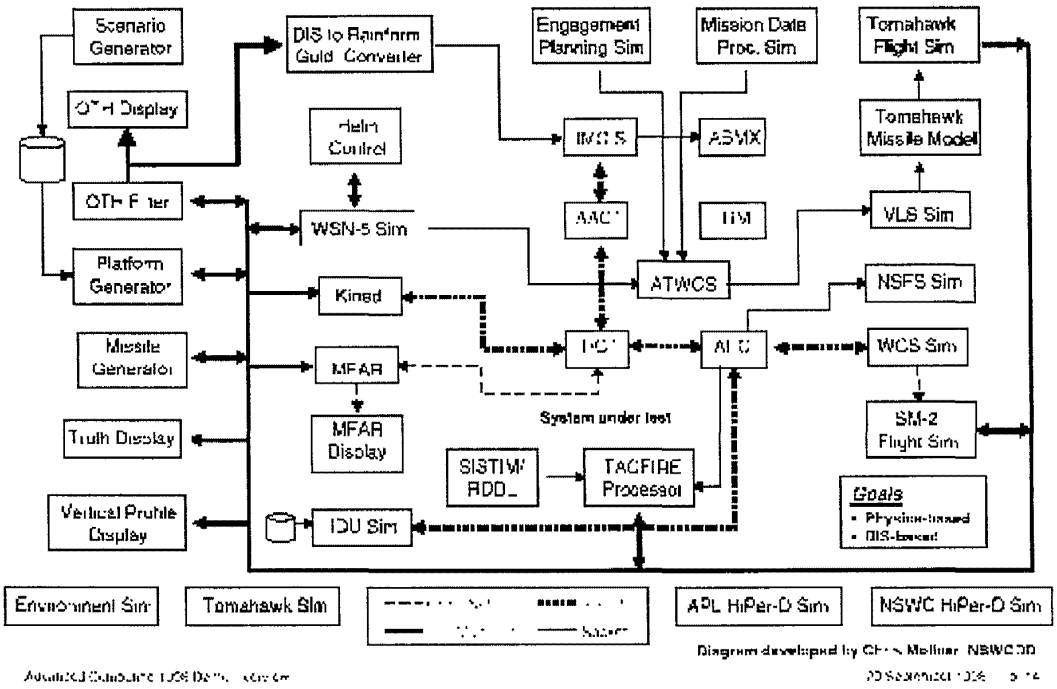


Figure 3.3.1-1. EnvSim Elements Using DIS

3.3.1.1 Entity Simulations

a. **Platform Generator**: The Platform Generator simulates all the background tracks in the scenario and initializes the NavSim. It reads in a data file specifying what platforms are to be simulated and their waypoints. During a scenario, a track follows the path defined by the waypoints. Tracks can perform the following actions:

(1) Change speed or course

(2) Change altitude

(3) Change IFF Modes

(4) Turn radars on or off

(5) Change radar modes

(6) Launch anti-ship missiles (ASMs)

If the appropriate command line argument is given to the Platform Generator at start up, the Platform Generator will generate a Set PDU and send it to the NavSim. The following data is included in the Set Data PDU:

(1) Force ID

(2) Entity ID

(3) Entity Kind

(4) Domain

(5) Country

(6) Category

(7) Subcategory

(8) Specific

(9) Extra

(10) Geocentric Coordinates – x, y, and z

(11) Velocity in Geocentric Coordinates

The NavSim in generating the ES PDU uses all the data in the Set Data PDU. The Platform Generator sends out the Set Data PDU once every 5 seconds until the NavSim responds to the Set Data PDU with an Acknowledge PDU.

b. **Missile Generator**: The Missile Generator simulates ASMs that have been launched at ownship. The ASMs use proportional navigation during flight and are of four types:

(1) Type I ASMs have a low cruise altitude and are subsonic

(2) Type II ASMs are sea-skimmers and subsonic

(3) Type III ASMs are high-divers and supersonic

(4) Type IV ASMs are sea-skimmers and supersonic

c. **NavSim & Helm Control**: Together, the NavSim and Helm Control represent ownship. The NavSim is based on a Tomahawk WSN-5 simulation that has been certified by Operational Test and Evaluation Force (OPTEVFOR). The NavSim generates position, velocity, and attitude data for ownship. The attitude data is sent to the NavSim Data Server at a 16 Hz rate. Position data is updated at a 1 Hz rate. The NavSim also generates an Entity State PDU for ownship.

The Helm Control simulates the ownship helm and is used to send helm commands to the NavSim. The commands are sent using a Set Data PDU. Figure 3.3.1.1-1 shows the Helm Panel Display.

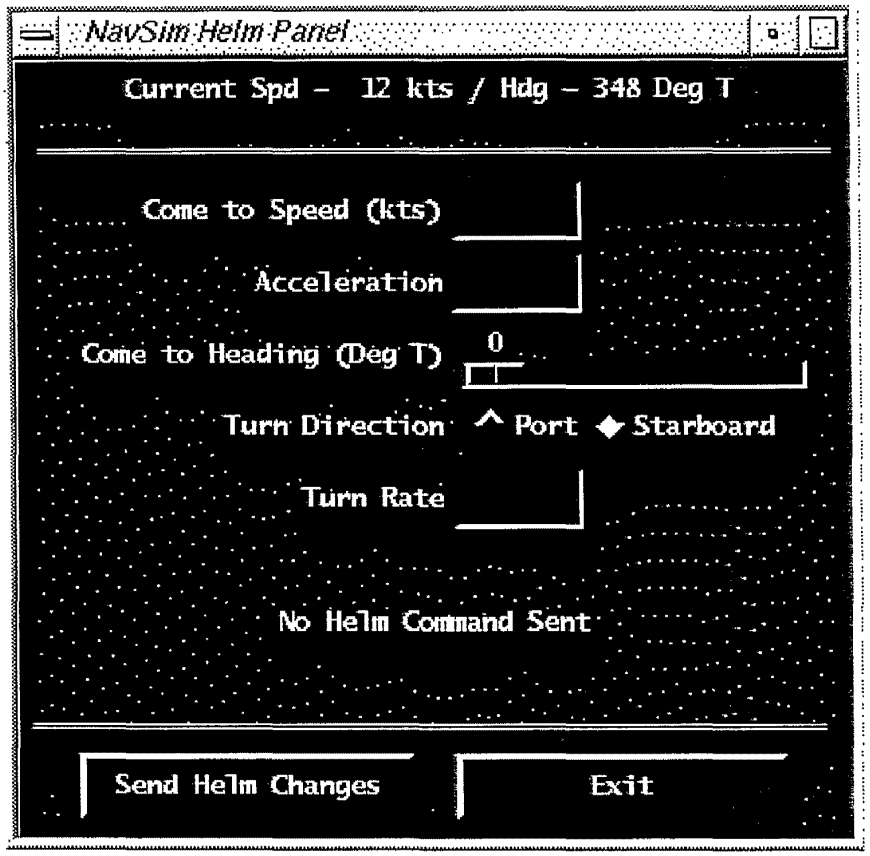


Figure 3.3.1.1-1 Helm Panel Display

d. **SM-2 Flight Sim:** The SM-2 Flight Sim simulates the SM-2 surface-to-air missile once it has been launched. The SM-2 Flight Sim receives launch commands from the AEC. After launch, the SM-2 Flight Sim generates an Entity State PDU during flight and uses proportional navigation to intercept the target.

e. **Tomahawk Flight Sim:** The Tomahawk Flight Sim simulates the Tomahawk Cruise Missile once it has been launched. The Tomahawk Flight Sim decouples the missile's flight into vertical and lateral equations of motion. The vertical equations of motion describe the missile's dynamics in the longitudinal plane as it transitions from its current inertial altitude above sea level to its commanded inertial altitude. The lateral equations of motion update the missile latitude and longitude as the missile navigates along the great circle path between waypoints.

The Tomahawk Flight Sim initiates modeling of the missile's trajectory at the end of boost conditions. The Tomahawk Flight Sim models the missile in one of 3 flight conditions: performing a waypoint turn, flying the great circle distance between waypoints, or performing a maximum acceleration pull-up terminal maneuver.

The Tomahawk Missile Model used in Demo 98 was written in C. It is currently in use by Scenario Generation and Reconstruction (SG&R), the embedded Tomahawk trainer in ATWCS, and by the Commanding Officers Simulated Tactical Display (COSTD), a BFTT-compatible submarine training system. The C missile model is derived from the Fortran Tomahawk missile model, which was certified by the Tomahawk Simulation Management Board.

3.3.1.2 Sensors Simulations

a. **MFAR:** The MFAR is the source of the real-time track data in the system. The MFAR sorts all DIS ES PDUs to determine potential tracks. The MFAR determines which tracks are detected based on radar horizon, sea clutter, standard atmospheric attenuation, specular multipath, and the track's radar cross section. Once a track is detected, it is passed on to the MFAR Broker.

b. **OTH Filter:** Together, the OTH Filter and DIS to Rainform Gold Converter are the sources of non-real-time tracks in the system. When the OTH Filter receives an ES PDU, the OTH Filter determines if the entity is ownship, an OTH sensor or an OTH track. If the entity is an OTH track, the OTH Filter determines if it is above the radar horizon of ownship and any OTH sensors. Once the identification is made, the OTH Filter determines whether to send, or not send, the ES PDU to the DIS to Rainform Gold Converter. Table 3.3.1.2-1 provides a criteria matrix for sending the Entity State PDU.

Table 3.3.1.2-1 Criteria to Pass Entity State PDU

OWNSHIP RADAR HORIZON	*OTH SENSOR RADAR HORIZON*	
	Above	Below
Above	Not Sent	Not Sent
Below	Sent	Not Sent

It should be pointed out that the OTH Filter is receiving ES PDUs on the same port as the Platform Generator, Missile Generator, MFAR, SM-2 Flight Sim, Tomahawk Flight Sim and NavSim. The ES PDUs sent to the DIS to Rainform Gold Converter are sent on a different port.

The TACFIRE Processor sends the OTH Filter an Event Report PDU containing data received from Remote Digital Data Link (RDDL) in a Call for Fire (CFF) Message. The OTH Filter uses the data in the Event Report PDU to generate an ES PDU which injects the CFF track into the system as an OTH track.

3.3.1.3 Displays

a. **Truth Display:** The Truth Display shows the location and speed of the tracks in the EnvSim, based on the ES PDU. The Truth Display operates in either absolute or relative mode. In absolute mode, the tracks are displayed on a grid based on their longitude and latitude. In relative mode, the tracks are displayed based on the range and bearing from a specified track (polar display). Examples of the absolute mode display and relative mode display are shown in Figures 3.3.1.3-1 and 3.3.1.3-2 respectively.

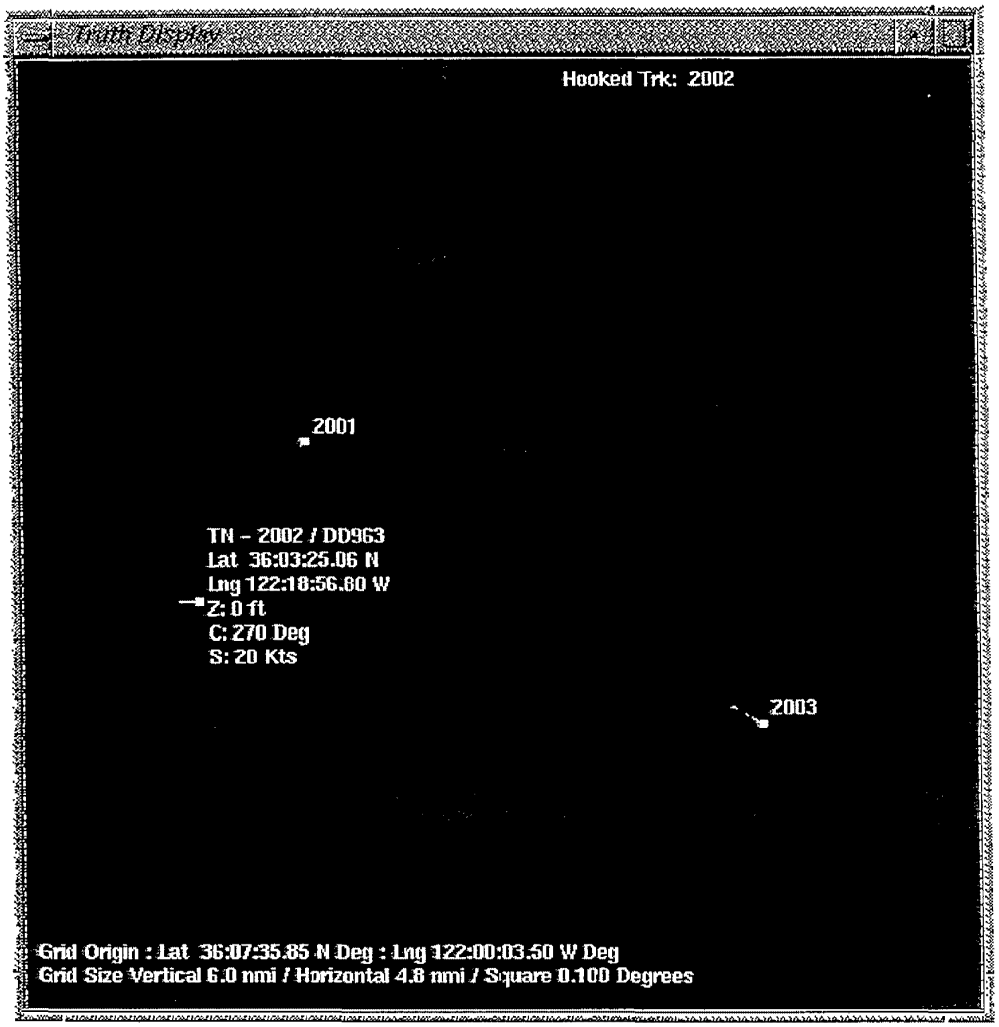


Figure 3.3.1.3-1 Truth Display in Absolute Mode

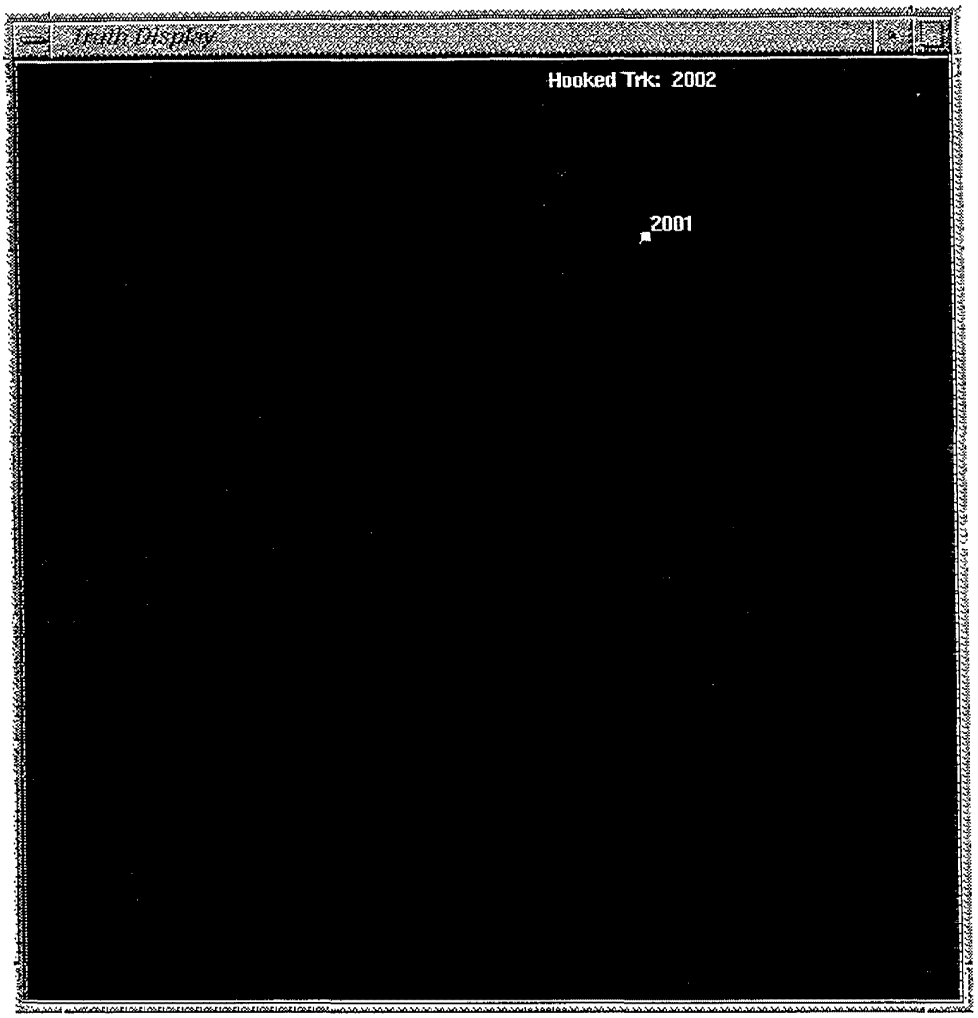


Figure 3.3.1.3-2 Truth Display in Relative Mode

b. **Vertical Profile Display:** The Vertical Profile Display displays the range and altitude of selected tracks from a specified track which, in this case, is ownship. The range from ownship is shown on the x-axis and the altitude above ownship is shown on the y-axis. The Vertical Profile Display uses the ES PDUs as its source of data. An example of the Vertical Profile Display is shown in Figure 3.3.1.3-3.

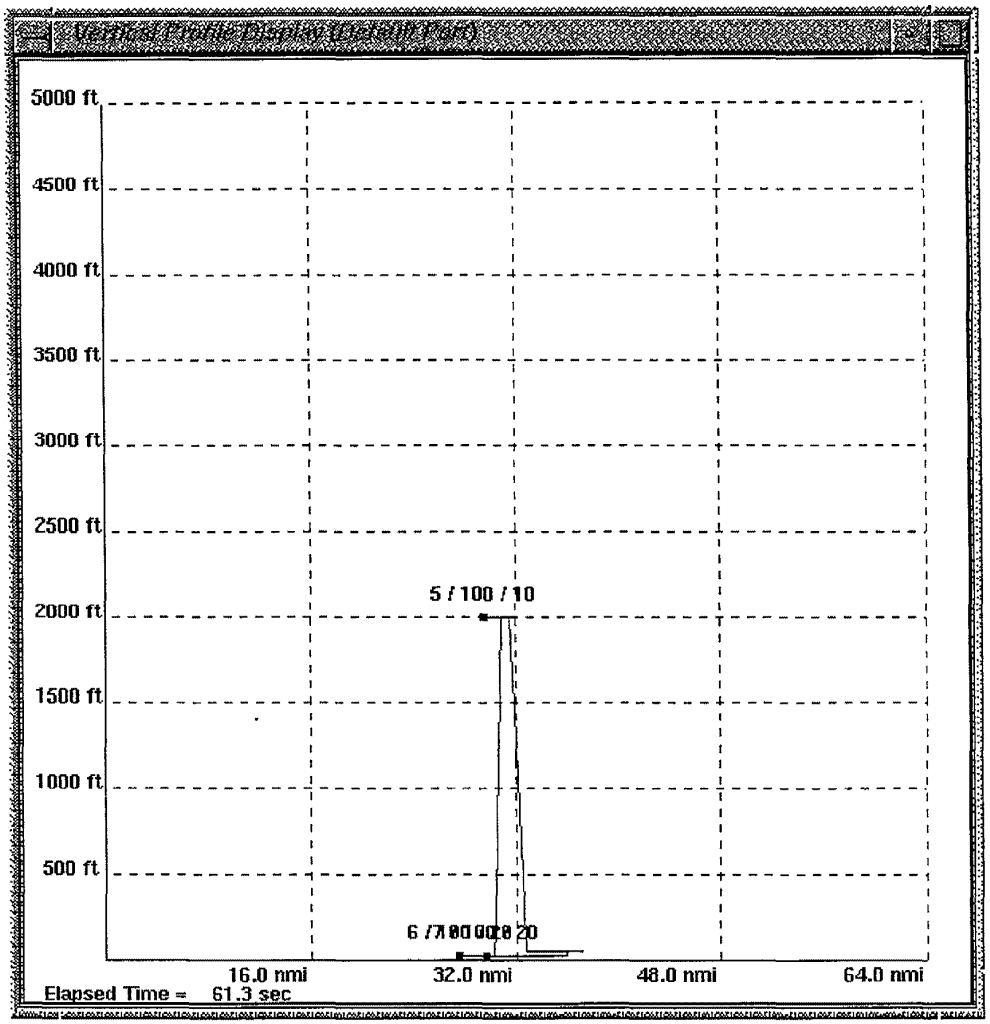


Figure 3.3.1.3-3 Vertical Profile Display

3.3.2 Simulation Control (SIMCON)

a. The SIMCON program performs two major functions. It reads a specially prepared[16] script file from disk that it uses to schedule creation of tracks and maneuvers according to absolute or relative timing information contained in the script file. It can also read DIS PDUs to accept a command to create a ring, or torus, of tracks. This PDU interface and functionality are new for Demo 98. Both modes of operation produce various HiPer-D internal messages that are broadcast to appropriate communications groups.

b. If SIMCON is to run only a script, it will terminate normally after it has finished all operations associated with the script. When SIMCON is to read PDUs, it will run indefinitely.

3.3.2.1 Modifications Description

a. Changes were made to the SIMCON program to allow it to create a ring of tracks in response to the newly defined DIS Ring of Death (RoD) Program Data Unit (PDU). Only the modifications to the functionality of SIMCON are described here. The new functionality is limited to the PDU interface and processing of a new type of PDU, the RoD, which is defined and used only by HiPer-D.

b. To receive PDUs, SIMCON creates a thread that opens a UDP socket and listens to a specified port and IP multicast group address. When it receives a message, it decodes the PDU header. If the PDU is a RoD PDU, it will decode the rest of the PDU. The RoD PDU is considered a request to generate a ring or torus of tracks. SIMCON will only handle such requests one at a time. The PDU interface thread will never exit.

c. To create the tracks described in the RoD PDU, SIMCON will create another thread that builds the data structures needed to create a new local track and puts them into the SIMCON list that holds time-ordered new local track requests. When the thread has created all its new local track requests and put them in the list, the thread exits normally.

d. If a second request arrives before processing of the first request (which includes passing information to SENSIM for all tracks associated with the first request), the second request will be rejected and never processed. A mutex is used to prevent simultaneous processing of multiple RoD requests.

3.3.2.2 Restrictions

a. When SIMCON is executed only to run a script, the program will terminate after it has fulfilled all the directives of the script. To run another script at a later time, one must run SIMCON again. This can be done without disrupting an ongoing HiPer-D run.

b. When SIMCON is instructed to read DIS PDUs it will not terminate; it will run indefinitely until aborted externally or through an error. It is possible for SIMCON to run a

[16] SIMCON reads a binary file that is created by running a Perl script, simcc, against an ASCII text file. This text file is in an easily read table format that defines new tracks, track drop, and track maneuver operations.

script and listen for PDUs, but this execution of SIMCON cannot subsequently be used to read another script. However, multiple copies of SIMCON can be run simultaneously so a second copy can be started to read a script file if desired.

3.3.3 Kinematics Daemon (KINED)

KINED generates periodic reports on tracks. It receives track initiation and destruction information from the SENSIM process and assigns unique numbers, obtained from the Track Number Server, to new tracks entered in the system. Once established, tracks are updated at a periodic rate specified by the Sensor Rate Server (SRS). KINED also inserts ownship data into the system. It does not, at this point, support any maneuvering of ownship. Multiple KINED processes may be run on different machines to distribute the load.

3.3.4 Weapons Control System Simulator (WCS Sim)

WCS Sim interfaces with the SENSIM to initiate and terminate engagements, and to receive notification of their completion. The WCS simulator randomly predetermines the success of the engagement and notifies SENSIM of this when the engagement is initiated. The success/failure is based upon a random number generator that results in an 85 percent probabilistic kill.

3.3.5 Identification Upgrade Simulator (IDU Sim)

IDU Sim provides identification information when it receives a quick identification request from the AEC engagement initiation processing. These requests are made upon track entry into the system and upon initiation of an engagement sequence for a particular target.

3.3.6 NSFS Simulator (NSFSsim)

a. NSFSsim interfaces with the Engagement Server and the C3I Broker processes in the AEC system as part of the land attack engagement sequence. It additionally receives OTH (Over-the-Horizon) track data from the OTH Data Server (OTHDS) that supplies positional information in support of ballistic trajectory calculations.

b. The Engagement Server delivers the initiating message of the land attack request to NSFSsim. This component then performs gun engagement checks and time-of-flight (ToF) calculations against the selected OTH target. NSFSsim then initiates a simulated land attack sequence by representing the firing of a 5" gun projectile via a Character-Readout (CRO) textual display to the operator. The primary purpose of this capability is to provide a basic approximation of the progress and status of the engagement sequence as might be witnessed at a prototyped operator position. Target data, a countdown ToF field, and engagement status data are represented in this output window. Subsequent spotter corrections are received by NSFSsim who recomputes the required parameters, issues the reply back to the C3I_Broker and commences follow-on firing sequences.

3.3.7 Digital Call For Fire Support Components

There are three support components that are part of the Digital Call For Fire capability. The Remote Digital Data Link (RDDL), the Tactical Fire Direction System (TACFIRE) Processor, and the C3I Broker.

3.3.7.1 Remote Digital Data Link (RDDL)

a. The RDDL simulates the Forward Observer/Forward Air Controller (FO/FAC) and communicates with the TACFIRE Processor via a TCP/IP connection using TACFIRE Fixed Formal messages.

b. To initiate a Digital Call For Fire sequence, RDDL sends a FR GRID message containing target position information. When the gun engagement is commenced and each time a round is fired, a "shot" message is received from the TACFIRE Processor to let the FO/FAC know that a round has been fired. Five seconds before round detonation, RDDL receives a "splash" message to cue the FO/FAC that impact and detonation is imminent. If the round is off target then RDDL will return a subsequent adjust (SUBQ_ADJ) message containing how to place the next round. Once the spotting round is on target, RDDL sends a SUBQ_ADJ message with the Fire For Effect bit set initiating several rounds from the ship onto the target. When the target is destroyed RDDL sends an EOM_SURV message declaring the target destroyed and the Call For Fire mission complete.

3.3.7.2 TACFIRE Processor

a. The TACFIRE Processor is a message protocol converter that converts the Fixed Format TACFIRE messages received from RDDL to the internal HiPer-D message format and converts HiPer-D message formats received from the C3I Broker into Fixed Format TACFIRE messages sent to RDDL. TCP/IP connections are used for both communication channels.

b. The TACFIRE Processor receives the FR_GRID message from the RDDL and uses the information to send an Engagement Request message to the C3I Broker. Each time a round is fired, a Time of Flight (TOF) message is received from the C3I Broker. The TACFIRE Processor then sends the "shot" message to RDDL and at the appropriate time (five seconds before impact) sends the "splash" message to RDDL. The TACFIRE processor receives the subsequent adjust message from RDDL, converts to HiPer-D message format, and transmits it to the C3I Broker. The same processing occurs with the Fire for Effect and End of Mission message.

3.3.7.3 C3I Broker

The C3I Broker provides ownship and Call For Fire request information to ABMX, ties the Call for Fire engagement request to an existing system track, passes the engagement information onto the Engagement Server, and returns Time of Flight messages back to the TACFIRE Processor.

3.3.8 System Control

a. System Control functions provide automated assistance in the start-up, restart, and monitoring of the ATCF processes that are distributed across the computing environment. These facilities include "agents" and "node managers" that reside on each machine, as well as a master "controller" that implements a planned configuration. There is also a simple shell-like language for describing the planned configuration (typically called "the plan").

b. On start-up, the controller reads and verifies the plan. It then establishes agents on each machine (via the node managers), and doles out the appropriate portion of the plan to each of the node managers. It then coordinates the overall start-up, assuring that applications dependent on others are started after the components upon which they depend. The controller also provides a simple user interface for monitoring system operation and effecting changes (such as shutdown or start-up of a component).

c. System Control provides for managing transitions that result from failed applications. On failure, an application can be automatically restarted. In cases where it cannot be replicated, and had dependent applications, those too would be automatically restarted. If however, it had a replica that did not fail, the failed application would be restarted with no action on the dependent components.

3.3.9 Clock Synchronization

a. A key to distributed processing approaches, and the use of COTS equipment in mission critical systems, is the coordination of all the individual clocks located in the system. A single time base is necessary for many of these applications and is required for instrumentation of a distributed system. In order to obtain a single stable time base, a time service needs to be provided. This time service includes the provision of time, the synchronization of time, and the management of the time service.

b. For this demonstration, the Network Time Protocol (NTP) was used to achieve synchronization of time. The NTP is a distributed clock synchronization protocol that provides for the coordination of interconnected computer clocks using the existing communication infrastructure. Dr. David Mills at the University of Delaware developed NTP for use by the Internet community. NTP calculates clock offsets between two peer clocks, and provides corrections to the appropriate clock system calls of the operating system. NTP uses a two-way exchange of time information to estimate the actual clock offset better. In addition, data filtering and clock selection algorithms are used to improve performance and stability.

c. For Demo 98, NTP Version 3 was installed on all platforms. The Alphas, Suns, HPs, and SGIs used the public domain version of XNTPD, the NTP daemon. A synchronization sub-net was constructed involving a single server with a backup. All other machines were clients to either the server or the backup.

3.3.10 Near Real-time Data Collection/Display (JEWEL)

a. JEWEL is a distributed measurement tool developed by the GMD National Research Center for Information Technology in Germany. JEWEL consists of a toolkit for low-interference on-line performance measurement, integrated with an adaptable graphical presentation facility, and a generic interactive experiment control system. Extraction points are inserted into the applications where measurements are to be made. When these extraction points are activated and executed, the appropriate data is collected, time-stamped, and placed in a shared memory buffer via the JEWEL internal sensor. The JEWEL external sensor retrieves the data and transfers it over the network (Ethernet used for this demonstration) to the JEWEL collector or Graphical Presentation System (GPS), where the data is then graphically displayed in a manner determined by the experimenter. The JEWEL instrumentation configuration is shown in Figure 3.3.10-1.

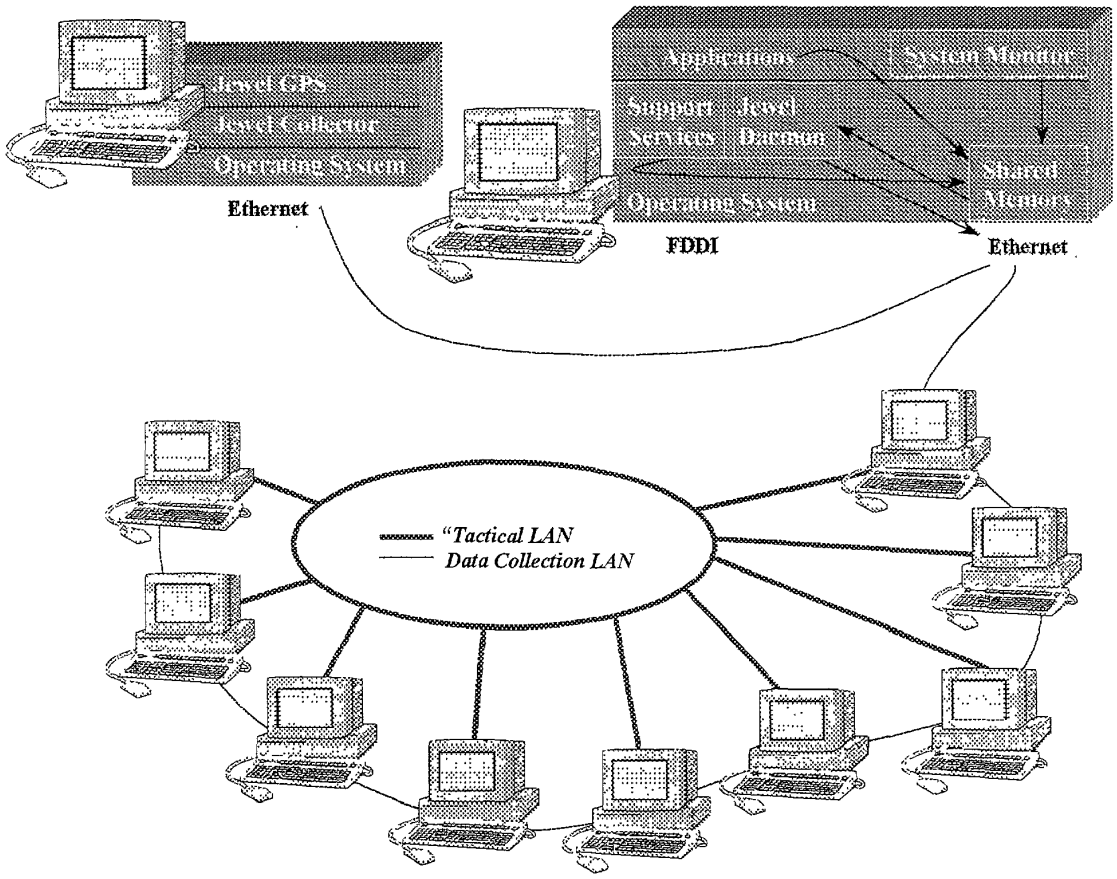


Figure 3.3.10-1 JEWEL Instrumentation Configuration

b. In order to provide robust instrumentation in the dynamic distributed environment of the testbed several significant improvements have been made to the Jewel package. The basic Jewel architecture is shown in Figure 3.3.10-2. There is a Jewel daemon component that resides on each host where an instrumented application might be placed. The Jewel daemon is

responsible for starting up any other Jewel components which need to run on the host. The component which sends control orders to the Jewel daemons is the Experiment Control System (ECS) component. The ECS component reads in a set of configuration files and then orders the startup of components as defined in these files. The key components that the Jewel daemon starts up are:

(1) The External Sensor which creates a shared memory queue for instrumentation events and reads events placed in the queue and forwards them to any of the other Jewel components which need the events.

(2) The Graphical Presentation System (GPS) components which start up and control the Jewel X/Motif instrumentation displays.

(3) The Collector components which serve as a broker interface for forwarding instrumentation events to external applications. (The Collector serves as the interface to the Resource Management components). In addition to these components, there is a library called the Internal Sensor which is linked into each instrumented application through which the application sends timestamped event data to the shared memory queue.

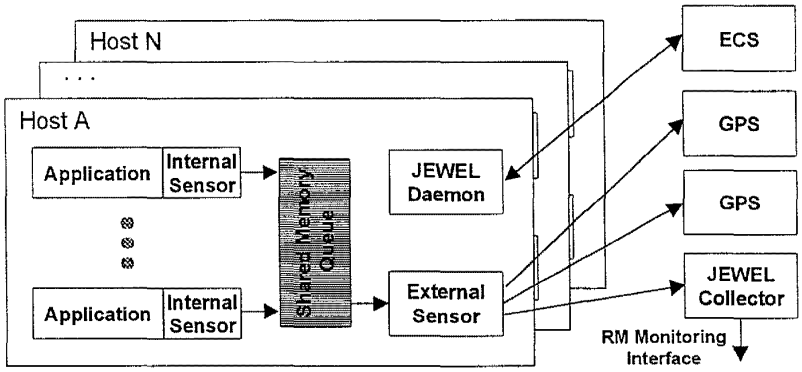


Figure 3.3.10-2. Jewel Instrumentation Architecture

c. The most significant improvement made was to the shared memory interface between the Internal Sensor and External Sensor. The original Jewel design required that all of the Jewel infrastructure components be started and running prior to the applications being run. This has been changed so that it does not matter whether the applications or the Jewel components are brought up first. Further, the Jewel components can also be brought up and down during a run with no impact on the behavior of the instrumented applications. The shared memory queue implementation has also been enhanced to allow larger message sizes and more messages to be stored. Much more reliable shared memory mutex capabilities have been implemented to ensure that readers and writers are not accessing the memory simultaneously. Also, the format of the instrumentation event messages have been enhanced to include the IP address of the host and to allow larger event message sizes.

d. The Collector component has also been improved to allow it to better serve as a pass-through for providing instrumentation events to external applications. To accomplish this, the XDR (External Data Representation) interface between the Collector and external components was considerably simplified. Also, an interface library was developed which allows a client to the Collector to request that only specific events are to be forwarded. This has greatly simplified

that the interface for receiving instrumentation events from Jewel and has created a "standardized" mechanism for requesting and receiving specific Jewel instrumentation events.

e. Also, changes have made to the Jewel Graphical Presentation System (GPS) X/Motif widget library which allows better support for dynamic allocation / reallocation of applications. This has been accomplished by adding features that allow the hostname and process id of an application to be retrieved and used within the widget library. For the current Jewel displays, this has allowed the legends to be updated to show the hostnames of the platforms where instrumented applications are currently running.

3.3.11 Group Communications

One of the core technologies on which HiPer-D is based is process group communications. Process group communications provide a mechanism in which applications become members of a communications group. When a member sends a message in a group, all other members of the group receive the message. In this respect, process group communications are analogous to multicast communications. Process group communications extend the concept of multicast by providing reliable communications, by guaranteeing different levels of message ordering, and by providing operations associated with membership changes in the group. See Appendix E for details of the group communications mechanism used in ATCF portion of HiPer-D for Demo 98.

3.4 Resource Management.

a. During FY98, the Resource Management capabilities within the HiPer-D testbed have been significantly expanded and enhanced. As shown in Figure 3.4-1, the scope of our Resource Management efforts are aimed at effectively monitoring and controlling the configuration of an interconnected shipboard network consisting of many general-purpose computing platforms. Within the HiPer-D testbed, the Resource Management components continuously monitor the state and performance of the system, determine when and if Quality of Service (QoS) requirements (real-time deadlines, desired throughput, fault tolerance, etc...) are not being met, and reallocate applications to resources as required in order to ensure that QoS requirements can be met. The goal is to allocate and, when necessary, reallocate combat system functions (i.e., applications) to computing resources in a manner that ensures that the mission-critical real-time requirements of the combat system are met. A paper presented at the 1999 IEEE Real-Time Technology and Application Symposium describing the Demo 98 Resource Management capabilities and results is attached as Appendix F.

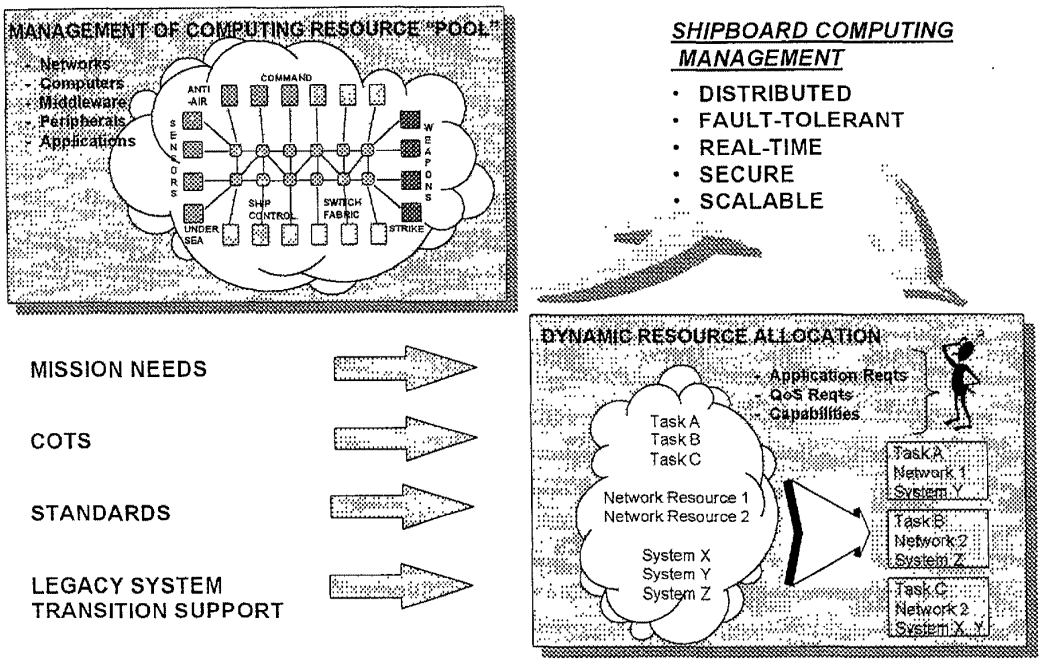


Figure 3.4-1. Resource Management Environment

b. The benefits of this approach include:

(1) It provides the ability to dynamically map resources based on changing mission requirements which permits flexibility during changing tactical situations (as in the case of transitioning from an open water environment into a complex, potentially hostile, littoral environment).

(2) It provides the ability to maintain mission capabilities in the event of equipment failure, software failure, damage, fire in spaces, flooding, or other catastrophic events.

(3) Perhaps most importantly, it provides the potential for significant life cycle cost savings and manning reductions; by moving away from statically configured stovepipe systems, system development, integration, and maintenance costs could be reduced, and by being able to reconfigure around failed equipment, at-sea maintenance costs (and the number of required maintenance personnel) could be lowered.

c. The Resource Management capabilities developed during FY98 were a joint effort between NSWCDD and Dr. Lonnie Welch at the University of Texas at Arlington (UTA) who was funded under the DARPA Quorum program. Figure 3.4-2 shows the scope of our Resource Management efforts.

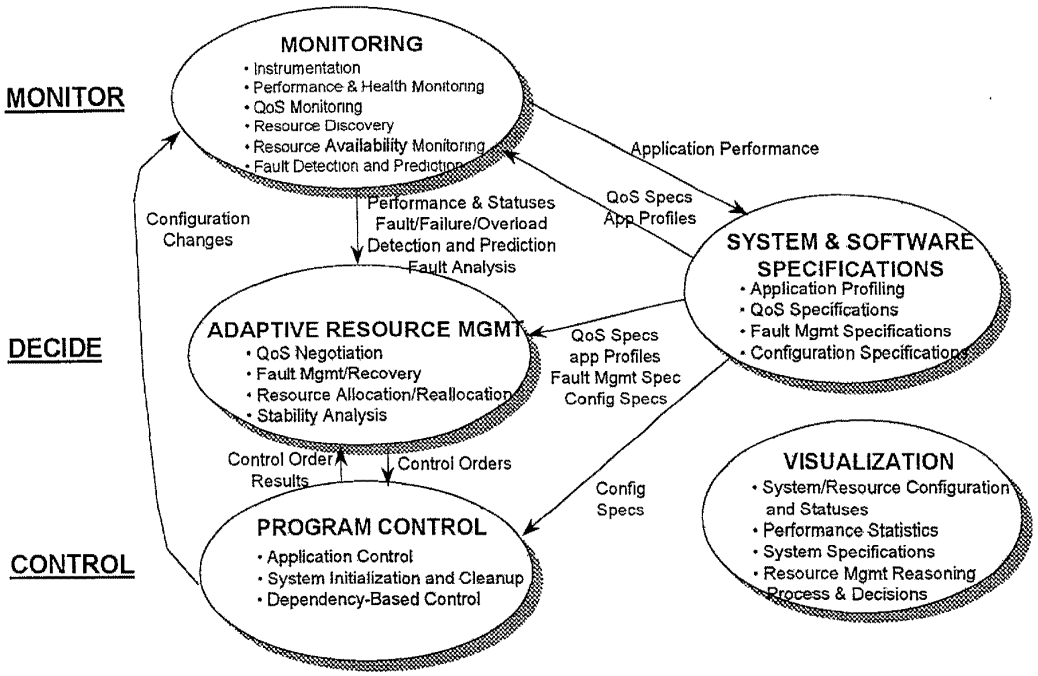


Figure 3.4-2. Resource Management Scope.

d. The major functions of resource management are to monitor, to analyze, and to adapt. The monitor function collects selected data concerning the behavioral and performance characteristics of all layers of the computer system, from the application layer through the computing and networking resources. The data collected during monitoring are analyzed based upon the system specifications to determine when computing resources should be reallocated. Reallocation will take place when a fault occurs or is predicted to occur. Reallocation will also take place when an unacceptable imbalance of application distribution occurs across the computing resources. The adapt function determines how to reallocate the application across the resources. This determination involves isolating the cause of the fault or overload, discovering available computing resources to overcome the fault or overload, and possibly, making tradeoffs in the quality of service (QoS) to be delivered to various tactical applications (based on their level of importance) in order to remedy the situation. A set of specifications are used to identify the processing capabilities of each component of the computing resource pool as well as the processing requirements of each of the applications and its level of importance. The Program Control components will carry out the reallocation of the applications across the computing resources. Also, a series of visualization displays were developed to enhance the monitoring, decision making, and control capabilities.

e. The Resource Management architecture is shown in Figure 3.4-3. Extensive upgrades were made to the Resource Management architecture and infrastructure which both provides new and enhanced resource management capabilities and also provides an infrastructure that can be built on and expanded in future years.

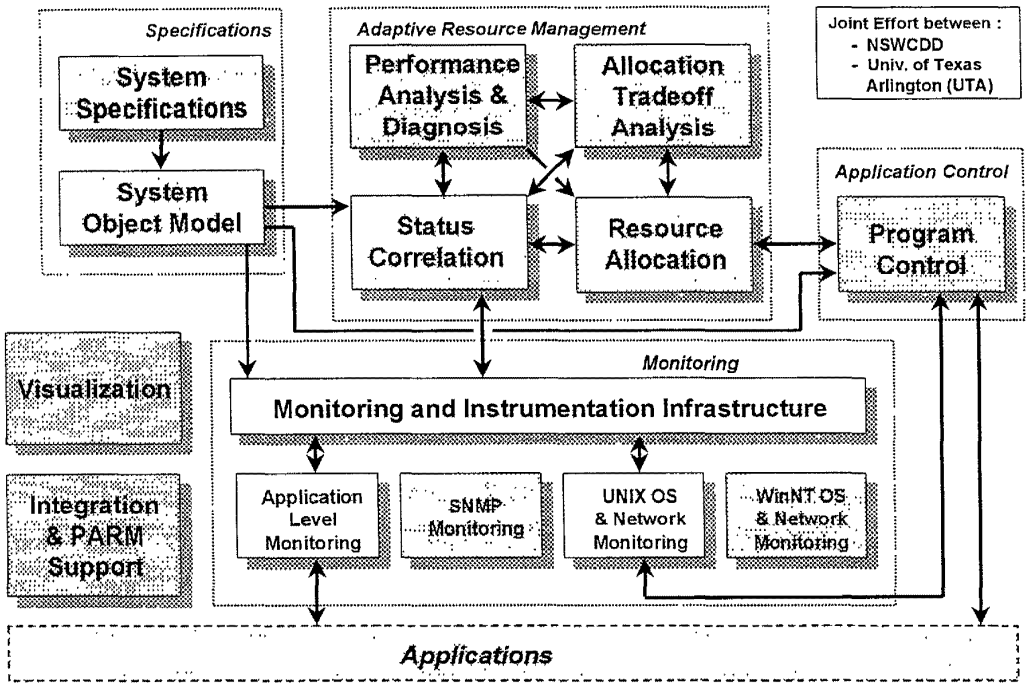


Figure 3.4-3 Resource Management Architecture.

f. The key focus areas have been Monitoring, Resource Management Decision Making, Control of the Applications, and the development of a QoS Specification grammar for defining system and application requirements.

g. At the Monitoring level, there are host monitors resident on each of the UNIX boxes to monitor operating system and network statistics. (Windows NT host monitors were also developed but were not fully integrated into the testbed during Demo 98; this is indicated by the gray shading in the architecture diagram.) For application-level instrumentation, a modified version of the Jewel instrumentation package was used. (Jewel was originally developed at the German National Research Center for Computer Science (GMD) in Germany.) Also, several initial tests were performed to attempt to determine whether SNMP (Simple Network Management Protocol) would be a viable candidate for real-time monitoring of operating system data. (The preliminary results have not been promising and these capabilities were not fully integrated during Demo 98.)

h. At the Decision Making level, the Resource Management components analyze the system performance, diagnose the cause(s) of poor performance, consider tradeoffs between potential resource allocations/reallocations when appropriate, and determine the best course of action to take to recover from fault or overload conditions.

i. At the Control level, the focus has been on providing application-level controls on the UNIX platforms. These capabilities include startup, shutdown, and configuration of both tactical and infrastructure applications on the UNIX platforms.

j. A fourth key area has been the development of a QoS and System Specification grammar that allows us to describe the capabilities and requirements of the software and hardware components of the testbed. This grammar allows us to define the structure of software systems and subsystems (e.g., AAW software components, ATWCS software components, Resource Management components, etc...). It also allows us to capture dependencies between components (such as startup order dependencies), information on how and where applications can be started and configured, and QoS requirements (e.g., timing requirements) either within an application or across an end-to-end path (consisting of multiple applications). The grammar also provides the ability to define host and network capabilities. The purpose of the grammar is to be able to precisely define the statically known structure, capabilities, and requirements of the system. This allows the Resource Management Monitoring, Decision Making, and Control components to use this information, along with dynamically monitored system status and performance data, to make effective decisions concerning the allocation of applications to computing resources.

k. As can be seen in the Resource Management architecture breakdown in Figures 3.4-3, the Monitor, Decide, and Control functions constitute a classic feedback control loop.

3.4.1 System Monitoring.

a. Extensive system monitoring capabilities have been developed and enhanced during FY98. The emphasis for FY98 has been the development of extensive monitoring and instrumentation capabilities at the application, host, and network levels which form an infrastructure that can be built on for future research.

b. The importance of robust, timely, and accurate monitoring at all levels in the system cannot be overemphasized. In order to make effective resource allocation decisions, it is imperative that the Resource Management components have access to accurate data on the status and performance of applications, middleware components, hosts, and networks.

c. The major monitoring and instrumentation efforts for FY98 are discussed in the remainder of this section. In particular, the major focus areas have been on UNIX OS and network monitoring, Windows NT OS and network monitoring, application-level instrumentation improvements, and status and history dissemination capabilities. A related effort is an ongoing study to determine whether and where current SNMP (Simple Network Management Protocol) standard MIB (Management Information Base) implementations can be used for real-time monitoring; the major issues being addressed include the overhead (in terms of CPU usage) of the SNMP queries and the latency and accuracy of the response data.

3.4.1.1 UNIX Operating System and Network Monitoring.

a. Early in FY98 it was determined that the various operating system status and performance monitoring techniques available from Network Management / System Management tool vendors were inadequate and/or inappropriate for use within the RM software infrastructure due to requirements for extensive monitoring, real-time performance, scalability, and minimal intrusiveness of the monitoring solution on the platform being monitored. What was needed was

a real-time system performance monitoring capability that could support a network-centric API (Applications Program Interface), as well as provide a system-wide user process monitoring/alarm capability. As a result of this need, a task was initiated to design, develop and implement a software suite, which provided the needed performance and process monitoring capabilities within the real-time requirements of the RM software framework.

b. One of the primary requirements was that the monitoring system software be capable of operating within a multi-platform environment while still providing a platform-independent API, along with a standardized data representation of system parameters and statistics. More specifically, the architectures and operating systems then under investigation were manufactured by Silicon Graphics, Sun Microsystems and Hewlett-Packard Corp., and consisted of vendor-specific Unix implementations based on the MIPS (IRIX 5.3, 6.3 and 6.5), SPARC (Solaris 2.5.1, 2.6 and 2.7) and PA-RISC (HPUX 10.20) machine architectures and operating systems. These goals were met with the release of SSMD (System Status Monitoring Daemon) which provided the requisite real-time performance statistics to the various software components of the RM software infrastructure.

3.4.1.1.1 Methodology.

a. The development of SSMD began with an analysis of the underlying architecture of whatever vendor-supplied system monitoring software was then currently available on the various platforms. This analysis led to the development of an initial list of critical system statistical parameters that SSMD would need to provide on a real-time basis. Once these parameters had been identified, the process of identifying and isolating these parameters within the various target Unix implementations began. This process consisted of an examination of each vendor's platform-specific system calls and their respective documentation. Where documentation was inconsistent or non-existent, experimental software needed to be written to allow the investigation of the actual behavior of the system call or parameter under investigation.

b. As a result of this investigation, a number of general categories detailing system parameters were isolated. These categories are as follows:

- *System-wide process parameters*
- **Overall system process-handling performance parameters. Some of these parameters are provided by the OS, and some are synthesized by SSMD during its data collection cycle.**
- *Per-process parameters*
- **Performance parameters of individual processes on a specific system. Some of these parameters are provided by the OS, and some are synthesized by SSMD during it's data collection cycle.**
- *System-wide CPU performance parameters*
- **The average CPU performance of the system. This usually consists of an average of the performance of all the CPUs within the system. This value may be provided by the OS, or synthesized by SSMD during its data collection cycle, depending on platform.**
- *Per-CPU performance parameters*

- **The performance of each individual CPU within the system. This will consist of items such as clock-tick and "percent" idle time, etc. The percentage based parameters are synthesized by SSMD during its data collection cycle.**
- *File system performance parameters*
- **A description of each file system mounted on the system. The description includes performance and utilization parameters for each individual file system.**
- *Network performance parameters*
- **A description of each active network interface on the system. The description includes performance and utilization parameters for each individual network interface.**
- *System Configuration parameters*
- **A description of the hardware and software components which make up the system, including items such as number of CPUs, CPU architecture, OS version numbers, etc.**

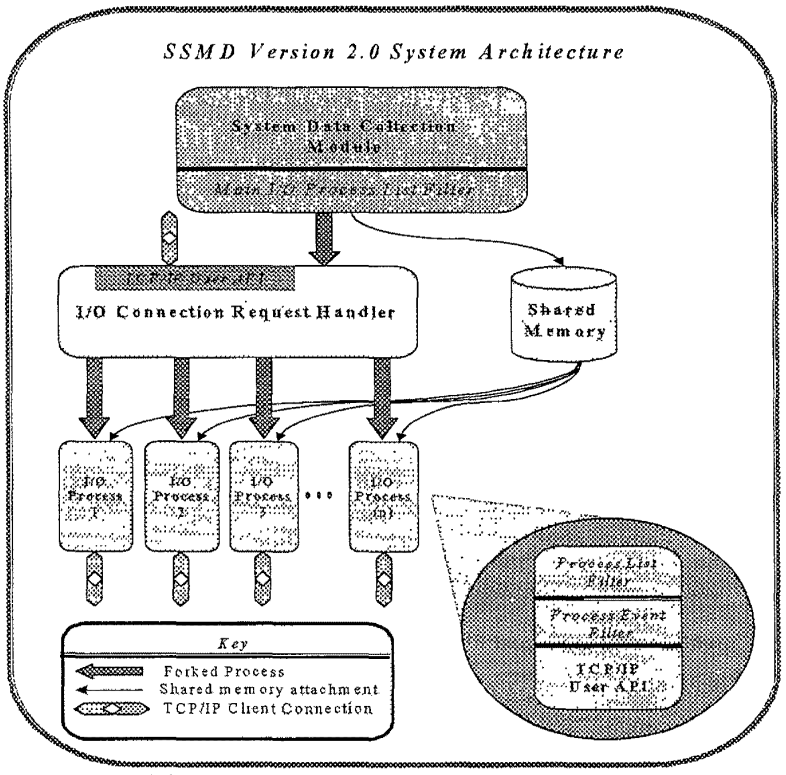


Figure 3.4.1.1.1-1. UNIX Host Monitor Design

c. A decision was made to build the SSMD server using an I/O model analogous to the standard network FTP server implementation. This I/O model, based on the Berkley Sockets model, would allow access to the SSMD server using a commonly available network communications API. The system statistics collection and distribution tasks were separated into two disparate processes, in order to minimize the impact of user API I/O on data collection timing. The resulting design is illustrated in figure 3.4.1.1.1-1. As can be seen from this

diagram, the SSMD server consists of 2 major processes, and an indeterminate number of I/O processes. This allows the server to handle large amounts of I/O in a scalable manner. Communications between the various processes within the server is handled via shared memory, allowing the use of an efficient publish/subscribe methodology. The shared memory interface also allows the server to be extended by allowing other platform-resident processes to attach to the shared memory area and gain access to system performance parameters without paying the performance penalty of using a sockets-based network interface.

d. Data collection timing is of critical importance. The server needed to provide system performance data to client applications on a real-time basis. As a result of this, SSMD was designed to allow collection, processing and distribution of system performance parameters based on a user-supplied variable timebase. During our initial tests, the timebase value was set at 2.0 seconds. Later in our testing phase, the timebase value was decreased to 1.0 second. This value provided performance data at a sufficient frequency to support the RM application software's needs at that time. The actual minimal SSMD timebase value, however, is much smaller, (and usually limited by platform dependent issues.) For the user, the limiting factor is the impact (or intrusiveness) of SSMD itself on system performance. With a data collection timebase value of 2.0 seconds, SSMD uses less than 1.0 % of CPU time on most platforms. With a timebase value of 1.0 second, the utilization is nominally less than 1.5%. If the user can tolerate higher SSMD CPU utilization, much smaller timebase values can be used.

3.4.1.2 Windows NT Operating System and Network Monitoring.

The Windows NT Host Monitor development was a joint effort between the University of Texas at Arlington (UTA) and NSWCDD. The NT Host Monitor software was written in Visual C++ by researchers at UTA as part of the Desiderata project which is funded under the DARPA Quorum project. The software was then modified by NSWCDD to add a graphical user interface and a TCP/IP network interface for remotely retrieving the machine statistics. Although the program was not used during the Demo 98 scenario, it was used for gathering information for characterizing the overhead involved in monitoring host and network statistics on the Windows NT platforms.

3.4.1.2.1 Windows NT Statistics Retrieval.

a. The NT Host Monitor is a Microsoft Windows NT 4.0 based program that provides a Windows graphical user interface to allow the user to select areas to monitor and to view the areas being monitored. Each area monitored has an icon associated with it. The user clicks an icon to start monitoring and displaying the data associated with that area. All areas can be monitored simultaneously or any subset of available data can be monitored. The icons act as a toggle switch; clicking the icon toggles between viewing the data associated with it and turning off the view. All data being displayed is updated at one-second intervals.

b. There are 5 main areas monitored by the NT Monitor. The first area is the processor metrics. It collects and displays all information associated with the time the processor has spent in each of the processor states (i.e., user mode, privileged mode, interrupt mode, and wait mode). It also collects the interrupts per second rate and displays it. The second area monitored is the

process list. It collects and displays all information associated with each of the processes residing on the system. It displays the amount of user mode and kernel mode CPU time used by the process, the amount of memory used by the process, and the ID of the process and image name including the path. The third area monitored is the system metrics. It monitors the data associated with the overall system such as file system statistics, CPU usage statistics, interrupts, and exceptions. The fourth area is the memory metrics. It monitors the overall usage of the memory such as memory used, memory available, paging statistics, cache memory usage, and memory available. The fifth area monitored is the network metrics. It monitors the overall network traffic. The network traffic is broken up into three categories: 1) network interface, 2) TCP interface, and 3) UDP interface. The network interface monitors the bytes sent and received and the packets sent and received. The TCP interface monitors connection statistics and the segments sent and received. The UDP interface monitors datagrams sent and received.

c. All data being gathered by the NT Monitor is collected from within the Windows NT Registry. The Registry data is maintained by the NT Operating System and updated at various rates by the Operating System. Our tests confirm that for the data being monitored and the polling rate currently being used (one second), the Registry is being updated frequently enough that problems with "stale data" have not been seen. Preliminary testing appears to indicate that monitoring of the system statuses and performance statistics on NT results in a much higher CPU load than for comparable system monitoring on Sun, SGI, and HP UNIX platforms. Initial tests shows CPU loads of 3 to 10% on the NT platforms as opposed to 0.2 to 1.5% for the UNIX platforms. However, additional testing is required before any definitive conclusions can be stated.

3.4.1.2.2 Network Interface.

The network interface to the NT Host Monitor was added by NSWCDD. The network interface is implemented as a TCP socket which listens for connections and sends out all currently monitored data at a one second rate when a remote connection is established. Currently, the format of the data being sent out is formatted ASCII strings which can be easily parsed by the receiver. The network interface can also be controlled via a tool bar entry that can be used to turn on or turn off the network connection.

3.4.1.3 Monitoring Status and History Servers.

a. The Monitoring Status and History Servers are data brokers between the host monitors and the other components of Resource Management. The Servers connect to the UNIX Host Monitors on the SGI, SUN, and HP platforms and collect and maintain both current and historical information about the systems, including processes running on the system, CPU information, network information, and file system information. The status and performance data and histories are provided to any of the Resource Management components (and potentially to other components) which need the data. In data is also filtered and reformatted as required by the client applications. Statuses and performance history data are currently being requested by and sent to the Resource QoS Monitor, the Host Display, the Graph Displays, and the Path Display.

b. In the current implementation, approximately 30 minutes of performance and status history data are maintained by the Servers for each monitored host in the testbed. For Demo98, there are a total of 37 hosts being monitored: 18 SUN Solaris 2.6 hosts, 2 SUN Solaris 2.5.1 hosts, 10 SGI IRIX 6.4 hosts, 1 SGI IRIX 6.3 host, 1 SGI IRIX 5.3 host, and 5 HP HP-UX 10.20 hosts.

3.4.2 Dynamic Resource Management.

a. The approach to adaptive resource and QoS management is based on the dynamic path paradigm. A path-based real-time subsystem 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. A system operates in an environment that is either deterministic, stochastic, or dynamic. A deterministic environment exhibits behavior that can be characterized by a constant value. A stochastic environment behaves in a manner that can be characterized by a statistical distribution. A dynamic environment (such as a war-fighting environment) depends on conditions which cannot be known in advance.

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

c. The approach described pertains to detection & assessment paths. This type of path *continuously* evaluates the elements of a sensor data stream to determine if environmental conditions are such that an action should be taken. Thus, this type of path is called *continuous.* Typically, there is a timeliness objective associated with completion of one review cycle of a continuous path, i.e., on the time to review all of the elements of one instance of a data stream. (The data stream is produced by sampling the environment. One set of samples is the data stream instance.)

d. The *threat detection* path of an air defense system is an example of a continuous path. It is a sensor-data-stream-driven path, with desired end-to-end cycle latencies for evaluation of radar track data. If it fails to meet the desired timeliness quality of service in a particular cycle, the path must continue to process track data, even though desired end-to-end latencies cannot be achieved. Peak loads cannot be known in advance for the *threat detection* path, since the maximum number of radar tracks that may exist in a battle environment cannot be known *a priori*. Furthermore, average loading of the path is not a useful metric, since the variability in the sensor data stream size is very large - it may consist of zero, 10s, 100s or 1000s of tracks.

3.4.2.1 System Model.

a. A demand space model based on the dynamic real-time path paradigm has been developed. A software subsystem, **SS**, consists of (1) a set of applications (**SS.A** = { $a_1, a_2, \ldots$ }), (2) a set of devices (sensors and actuators) (**SS.D** = { $d_1, d_2, \ldots$ }), (3) a communication graph defining the connectivity between applications and devices ($\Gamma(\mathbf{SS}) \in \Pi((\mathbf{SS.D} \cup \mathbf{SS.A}) \times (\mathbf{SS.D} \cup \mathbf{SS.A}))$), and (4) a set of paths (**SS.P** = $\{P_1, P_2, P_3, \ldots\}$). (Note: Π denotes the power set).

b. Each *continuous path* $\mathbf{P_i}$ is represented as (1) a set of applications $\mathbf{P_i.A} = \{a_{i,1}, a_{i,2}, \ldots\}$ (where $\mathbf{P_i.A} \subseteq \mathbf{SS(P_i).A}$), (2) a set of devices $\mathbf{P_i.D} = \{d_{i,1}, d_{i,2}, \ldots\}$ (where $\mathbf{P_i.D} \subseteq \mathbf{SS(P_i).D}$), (3) a communication graph $\gamma(\mathbf{P_i}) \in \Pi((\mathbf{P_i.D} \cup \mathbf{P_i.A}) \times (\mathbf{P_i.D} \cup \mathbf{P_i.A}))$ (note that $\gamma(\mathbf{P_i}) \subseteq \Gamma(\mathbf{SS(P_i)})$), and (4) a data stream $\mathbf{P_i.DS}$. (Note: $\mathbf{SS(P_i)}$ denotes the subsystem in which path $\mathbf{P_i}$ is contained.) $\mathbf{Profile(a_i)}$ is the set of hosts where application 'a_i' is eligible to be run (i.e., the set of hosts for which a_i has been compiled). For the communication graph $\gamma(\mathbf{P_i})$, the head node of the graph (which is the application which receives the initial input data stream) is represented as $\mathbf{ROOT(P_i)}$, and the last node of the graph (which is the application which communicates with other applications or paths outside of $\mathbf{P_i}$) is represented as $\mathbf{SINK(P_i)}$. The type of $\mathbf{P_i}$'s data stream is defined as $\tau(\mathbf{P_i.DS}) \in$ {dynamic, stochastic, deterministic}. (For the remainder of this paper, it is assumed that the all data stream types are dynamic).

c. The real-time QoS requirements of a continuous path include one or more of the following: (1) required latency of $\lambda_{\mathbf{REQ}}(\mathbf{P_i})$ seconds, (2) required throughput of $\theta_{\mathbf{REQ}}(\mathbf{P_i})$ data stream elements per second, and (3) required data inter-processing time of $\delta_{\mathbf{REQ}}(\mathbf{P_i})$ seconds (the maximum allowable time between processing of a particular element of $\mathbf{P_i.DS}$ in successive cycles). To mask transient QoS violations during QoS monitoring, a specification may also define a sampling window and a maximum number of QoS violations to be tolerated within the window; $\omega(\mathbf{P_i})$ models the sampling window size and $\upsilon(\mathbf{P_i})$ represents the maximum allowable number of violations within the sampling window.

d. The demand space model also captures information that must be obtained *a posteriori*. Some application programs can be replicated for load sharing. The set of replicas of application '$a_{i,j}$' during cycle 'c' of $\mathbf{P_i}$ is defined as $\mathbf{REPLICAS}\ (\mathbf{a_{i,j},c}) = \{\mathbf{a_{i,j,1}}, \mathbf{a_{i,j,2}}, \ldots\}$. The host to which application '$a_{i,j,k}$' is assigned during cycle 'c' of path $\mathbf{P_i}$ is defined as $\mathbf{HOST}\ (\mathbf{a_{i,j,k},c,P_i})$.

e. The set of elements that constitutes a data stream can vary dynamically. $\mathbf{P_i.DS(c)}=\{\mathbf{P_i.DS(c)}_1, \mathbf{P_i.DS(c)}_2, \ldots\}$ represents the set of elements in $\mathbf{P_i.DS}$ during cycle 'c' of $\mathbf{P_i}$. The **tactical load** (in number of data stream elements processed) of a continuous path $\mathbf{P_i}$ during it's $c^{\underline{th}}$ cycle is $|\mathbf{P_i.DS(c)}|$. The processing of elements of a data stream may be divided among replicas of an application to exploit concurrency as a means of decreasing execution latency of a path. In successive stages of a path that has non-combining applications (applications which, after processing data received from a single predecessor, simply divide the data among their successors), data will arrive in batches to applications; hence, each application may process several batches of data during a single cycle. Thus, the model represents the set of elements from *all* batches of data processed by application/replica 'a' during cycle 'c' as $P_i.DS(c, a)=\{P_i.DS(c, a)_1, P_i.DS(c, a)_2, \ldots\}$ The cardinality $|P_i.DS(c, a)|$ is the tactical load of 'a' in cycle 'c'. The data

stream elements contained in the jth batch of 'a' are denoted by $P_i.DS(c, a, j)=\{P_i.DS(c, a, j)_1, P_i.DS(c, a, j)_2, \ldots\}$.

3.4.2.2 Adaptive QoS and Resource Management.

This section defines metrics and techniques for reasoning about the mapping of demand space onto supply space, i.e., for resource and QoS management. The approach (depicted in Figure 3.4.2.2-1) works as follows. Application programs of real-time control paths send time-stamped events, via the *Application Instrumentation* component, to the *Path QoS Monitor* component. The *Path QoS Monitor* component calculates path- and application-level QoS metrics, compares observed QoS to required QoS, and notifies the *QoS Diagnosis* component when QoS violations are detected. The *Host & Network Monitoring* component collects operating system and network performance, status, and load information, which is then provided to the *Resource QoS Monitor* component. Here, host and network statistics are correlated, performance and load histories are maintained, and load metrics are calculated. This information is made available to the *QoS Diagnosis* component for use in determining resource loading, and allocation tradeoffs. The *QoS Diagnosis* component determines the cause of QoS violations, analyzes and ranks potential reallocation actions for restoring required QoS, and provides this list of recommended actions along with associated host and network load metrics to the *Resource Allocation* component. The *Resource Allocation* component determines the most beneficial allocation of resources for restoring required QoS. The allocation actions selected are then implemented by the *Application & Resource Control* components. The major components in Figure 3.4.2.2-1 are explained in more detail in the remainder this section.

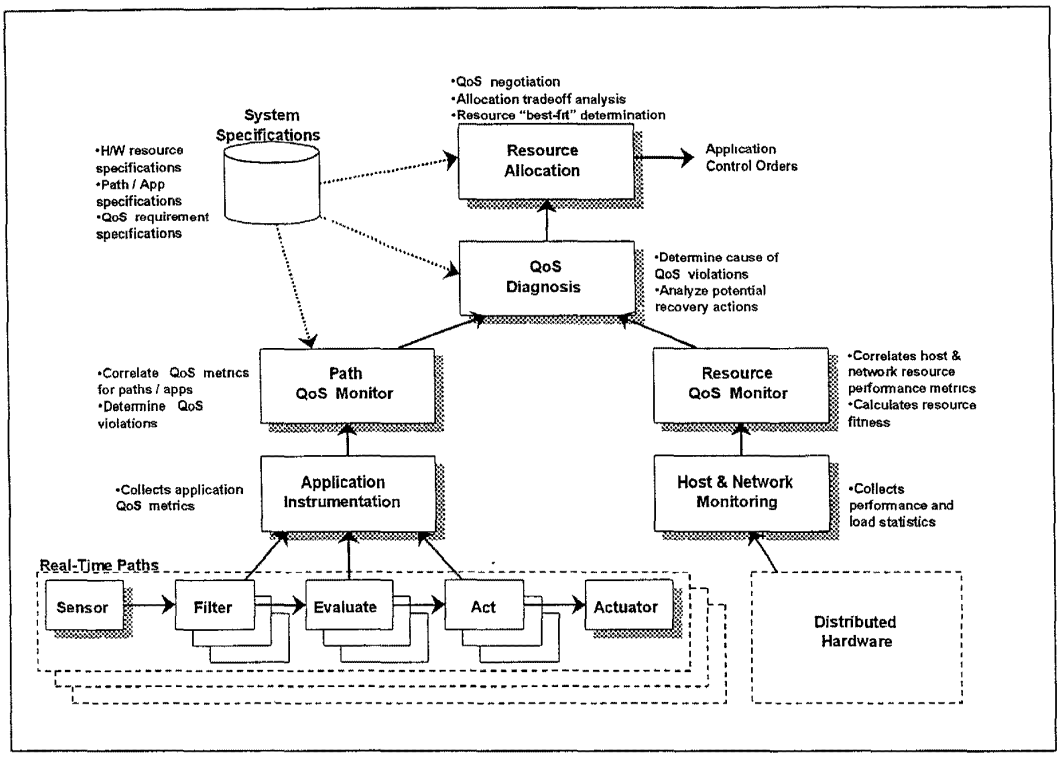


Figure 3.4.2.2-1 QoS and Resource Management

3.4.2.2.1 Path QoS Monitor.

a. The *Path QoS Monitor* component works as follows. Monitoring of real-time QoS involves the collection of time stamped events sent from applications. The times when application/replica 'a' starts and ends processing of the data stream for cycle 'c' are represented as $\mathbf{s(P_i.DS(c, a))}$ and $\mathbf{e(P_i.DS(c, a))}$, respectively. The times when application/replica 'a' starts and ends processing batch 'j' of data during cycle 'c' of P_i are denoted by $\mathbf{s(P_i.DS(c, a, j))}$ and $\mathbf{e(P_i.DS(c, a, j))}$, respectively.

b. Observed real-time QoS metrics are defined in terms of these basic events as follows: (1) <u>latency of path $\mathbf{P_i}$</u> during cycle 'c' is $= \lambda_{OBS}(\mathbf{P_i,c}) = \mathbf{max(\{e(P_i.DS(c, a_{i,m,n}, j)) - s(P_i.DS(c, a_{i,x,1}, 1)) \mid a_{i,m} = SINK(P_i), a_{i,x} = ROOT(P_i)\})}$ (note that λ_{OBS} is the maximum value from the *set* of latencies of all batches of data processed by all replicas of $\mathbf{SINK(P_i)}$ during the cycle), (2) <u>data-inter-processing time of application 'a'</u> in path $\mathbf{P_i}$ during cycle 'c' of data stream $\mathbf{P_i.DS(c,a)}$ is approximated as $\delta_{OBS}(\mathbf{P_i.DS(c,a)}) = \{\mathbf{s(P_i.DS(c,a)) - s(P_i.DS(c-1,a))}\}$, for $c > 1$, (3) <u>data-inter-processing time of path $\mathbf{P_i}$</u> during cycle 'c' for data stream $\mathbf{P_i.DS(c,a)}$ is $\delta_{OBS}(\mathbf{P_i.DS(c)}) = \delta_{OBS}(\mathbf{P_i.DS(c,a)})$, where 'a' = $\mathbf{ROOT(P_i)}$, (4) <u>observed cycle throughput of path $\mathbf{P_i}$</u> during cycle 'c' is $\theta_{OBS}(\mathbf{P_i,c}) = |\mathbf{P_i.DS(c)}| / \lambda_{OBS}(\mathbf{P_i,c})$, (5) <u>workload of application/replica 'a'</u> of path $\mathbf{P_i}$ during cycle 'c' is $\mathbf{W_{OBS}(P_i,c,a)} = |\mathbf{P_i.DS(c,a)}| / \delta_{OBS}(\mathbf{P_i.DS(c,a)})$, and (6) <u>workload of path $\mathbf{P_i}$</u> during cycle 'c' is $\mathbf{W_{OBS}(P_i,c)} = |\mathbf{P_i.DS(c)}| / \delta_{OBS}(\mathbf{P_i.DS(c)}) = (\Sigma|\mathbf{P_i.DS(c,a_{i1k})}|) / \delta_{OBS}(\mathbf{P_i.DS(c)})$, for all replicas *k* of $\mathbf{ROOT(P_i)}$.

c. Analysis of a time series of the real-time QoS metrics enables detection of QoS violations. An **overload** of a path or application occurs in any cycle 'c' where the number of violations within the sample window $\omega(\mathbf{P_i})$ equals or exceeds the maximum number of violations $\upsilon(\mathbf{P_i})$. As an example, detection of a path-level QoS latency violation occurs when the observed path latency $\lambda_{OBS}(\mathbf{P_i})$ exceeds the required path latency $\lambda_{REQ}(\mathbf{P_i})$ for $\upsilon(\mathbf{P_i})$ samples within the sample window of the most recent $\omega(\mathbf{P_i})$ samples. This can be expressed as $\upsilon(\mathbf{P_i}) \leq |\{d: (c-d)+1 < \omega(\mathbf{P_i}) \wedge [(\lambda_{REQ}(\mathbf{P_i}) - \lambda_{OBS}(\mathbf{P_i,d})) < 0)]\}|$, where 'c' is the current data stream cycle and 'd' represents data stream cycles within the sliding window $[c-(\omega(\mathbf{P_i})-1), c]$. For the experiments described in the subsequent section, path latencies ($\lambda_{OBS}(\mathbf{P_i, c})$) are used for determining QoS violations.

3.4.2.2.2 QoS Diagnosis.

The *QoS Diagnosis* component works as follows. When a path-level (end-to-end) real-time QoS violation occurs, diagnosis determine the cause(s) of the violation (i.e., identifies subpaths (application programs) that are experiencing significant slowdown). One diagnosis technique declares an application/replica 'a' to be unhealthy during cycle 'c' of path P_i if there exists another cycle 'd' such that the following conditions hold. **Condition 1:** $d < c$. **Condition 2:** HOST (a, c, P_i) = HOST (a, d, P_i). **Condition 3:** $|P_i.DS(c, a)| = |P_i.DS(d, a)|$. **Condition 4:** $\forall f$: $(f < c) \wedge [\text{HOST}(a, c, P_i) = \text{HOST}(a, f, P_i)] \wedge [|P_i.DS(c, a)| = |P_i.DS(f, a)|] \wedge \max(\lambda_{OBS}(P_i,f)) > \max(\lambda_{OBS}(P_i,d))$. **Condition 5:** $\max(\lambda_{OBS}(P_i,d)) < \max(\lambda_{OBS}(P_i,c)) - \varepsilon$. Note: ε is the minimal difference between cycle latencies that is considered significant.

3.4.2.2.3 Resource QoS Monitor.

a. The *Host & Network Monitoring* component and the *Resource QoS Monitor* component model the supply space *a posteriori* as follows. A hardware system, **HS**, consists of (1) a set of hosts **HS.H** = $\{h_1, h_2, \ldots\ldots\}$, (2) a set of Local Area Networks or LANs, **HS.L** = $\{L_1,L_2, \ldots\ldots\}$, and (3) a set of interconnecting devices **HS.I** = $\{i_1,i_2, \ldots\ldots\}$. The system model captures several hardware load metrics. The *paging score* of a host h_i at time t is defined as $\mathbf{PS(h_i,t)}$, and is calculated as the number of page faults per second averaged over the time interval t_1, divided by a maximum page fault threshold. The *cpu score* of a host h_i at time t is defined as $\mathbf{CS(h_i,t)}$, and represents the average percent CPU idle time over time interval t_2. The *network score* of a host h_i at time t is defined as $\mathbf{NS(h_i,t)}$, and is calculated as the number of packets received plus the number of packets sent averaged over time interval t_3, divided by a maximum network packet threshold. (All scores fall within the interval [0,1].)

b. Fitness scores for each of the host load metrics are calculated as follows: The *paging fitness* is calculated as $\mathbf{PF(h_i,t)} = (1 - \mathbf{PS(h_i,t)})$. The *cpu fitness* is calculated as $\mathbf{CF(h_i,t)} = \mathbf{CS(h_i,t)}$. The *network fitness* is calculated as $\mathbf{NF(h_i,t)} = (1 - \mathbf{NS(h_i,t)})$. These fitness score are used to calculate the **aggregate fitness indices**. The notation $\mathbf{FI(h_i,t)}$ denotes the *aggregate fitness index* of host h_i at time t. One fitness index function that we have found useful is: $\mathbf{FI(h_i,t)} = (\mathbf{w_1} * \mathbf{PF(h_i,t)}) + (\mathbf{w_2} * \mathbf{CF(h_i,t)}) + (\mathbf{w_3} * \mathbf{NF(h_i,t)})$, where w_i is the weight given to the ith load metric, and $\Sigma\, w_i = 1.0$. The fitness index is a relative measure of host load: the higher the fitness

index, the lighter the load on the host. When making resource allocation decisions, hosts with higher fitness scores are preferred over hosts with lower fitness scores.

3.4.2.2.4 Resource Allocation.

The *Resource Allocation* component works as follows. Its kernel is the best-host algorithm, which determines the "best" host on which to re-start or scale a candidate application. The algorithm only considers the set of hosts that are *eligible*. An eligible host is a host where an application is prepared for execution. The set of eligible hosts of each application is obtained from the system specification. The best host is determined using a "fitness" function. The host fitness index function used is $\mathbf{FI(h_i,t) = (w_1 * PF(h_i,t)) + (w_2 * CF(h_i,t)) + (w_3 * NF(h_i,t))}$.

3.4.2.3 Results.

a. The ability of the RM components to provide survivability services to real-time application systems in a timely manner was tested. In these tests, one replica of the AutoSpecial application was faulted, requiring that the RM compoments

(1) detect the failure

(2) restart a replica on the "fittest" of the eligible hosts.

b. These tests were performed a total of 17 times, and the reallocation decision times and total recovery times were measured. The average resource allocation decision time was 0.00097059 seconds, with a standard deviation of 0.00041648. The minimum, average and maximum total latencies of the recovery actions were 0.1296, 0.19401765, and 0.2379 seconds, respectively, with a standard deviation of 0.04376704. Thus, across all tests, the total response time for application fault detection and recovery services was far less than one second, providing adequate response times.

3.4.3 Resource Control / Program Control.

The three major areas of Dynamic Resource Management are monitoring, decision-making, and control. Program Control is the Resource Management "control" solution that provides a mechanism to change the status of a software system and the power to reach into a resource pool of many hosts and processors. To be put in perspective, Demo98 contained approximately 150 applications started through Program Control distributed across eleven Silicon Graphics workstations, eighteen Sun workstations, and five Hewlett-Packard workstations. In addition, Program Control is the entry-point for human-operators and automated functions, such as the resource-manager, for interaction with the computing-plant. The architecture is broken down into three major components:

3.4.3.1 Graphical User Interface.

a. The concept behind the Program Control interface is to provide a console that operators can log in to and gain access to a resource pool for startup and shutdown of any number of applications with a simple point and click. The display can create new configurations or continue pre-saved configurations. To create a new configuration, the operator can choose

from existing systems and components already defined in the System / QoS Specifications files or create their own entries. Figure 3.4.3.1-1 shows a typical edit window for selecting and customizing applications for a configuration. Once a configuration is built, the operator can save the startup order, static host-allocations, command-line arguments, etc... to a file for future runs. Since configuration files are ascii-text, users can easily create their own configuration files offline. Here is a configuration file example for the AutoSpecial doctrines used in Demo98:

Application	AAW:Doctrine:Auto_Special(1)
Host	altair4
Auto_Start	1
RM_Start	0
Directory	"$HIPERD_AAW_VERSION/exes"
Startup	"auto_special.solaris2.6.exe"
Time_Delay	2
Args	"%(UNIQUE, 1, 32) A_Spcl_%(UNIQUE, 1, 32) -jewel -rstat –splot "
Process	"auto_special.solaris2.6.exe"
Shutdown	SIGKILL

b. A description of each field is as follows:

Application:	Unique application name
Host:	Static host allocation
Auto_Start:	Should this application be part of a one-button startup
RM_Start:	Should the host be dynamically allocated
Directory:	Run-time working directory
Startup:	Name of application binary/script
Time_Delay:	Number of seconds to delay before starting this application
Args:	Command-line arguments to pass to the startup binary/script
Process:	List of processes expected to be seen by this application
Shutdown:	Name of kill signal or script to shutdown this application

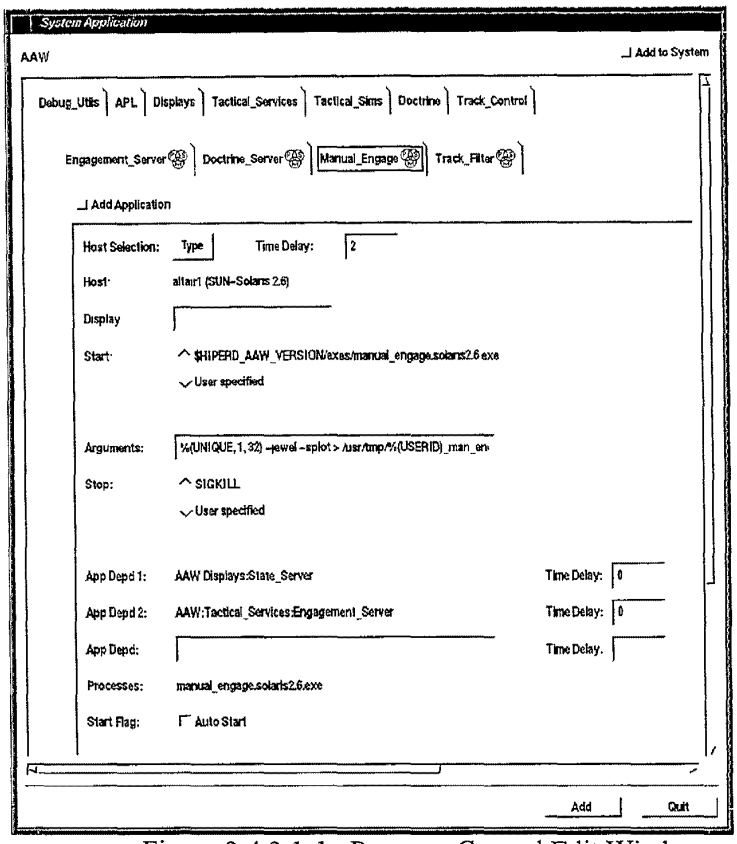


Figure 3.4.3.1-1. Program Control Edit Window.

c. Host selection/allocation can be done manually by the operator or automated by the resource-manager by simply changing a flag in an applications configuration. For manual operation, the user is given a list of "eligible" hosts to choose from. For a host to be "eligible", it must satisfy the conditions set forth in the System Specifications file. Of course, if the application isn't pre-defined in the specs, the operator must choose from the entire resource pool. Dynamic, or automated, host allocations are performed by the resource-manager upon the application start request being initiated.

d. Before any applications can be started, the rest of the Program Control structure must be started. Therefore, the operator presses the "Start Managers" button on the display. This causes hosts to be selected from the Hardware Specifications file to run managers. Currently, each manager will handle up to ten hosts, or agents. Once the managers are started, the operator presses the "Start Agents" button on the display to send a request to each manager to start its corresponding agents.

e. The display then provides a "view" into the computing plant by showing the status of applications: running, failed, stopped, or simply waiting to be started. Since no one view satisfies all operators, three view options are available: system-level view, application run-order view, and host view. The first displays applications associated with their relative systems and

subsystems; refer to figure 3.4.3.1-2. The second shows the order that applications should and will be started based on dependencies specified in the System Specifications; figure 3.4.3.1-3. And the last shows applications associated with the host they are/were/will be running on; figure 3.4.3.1-4.

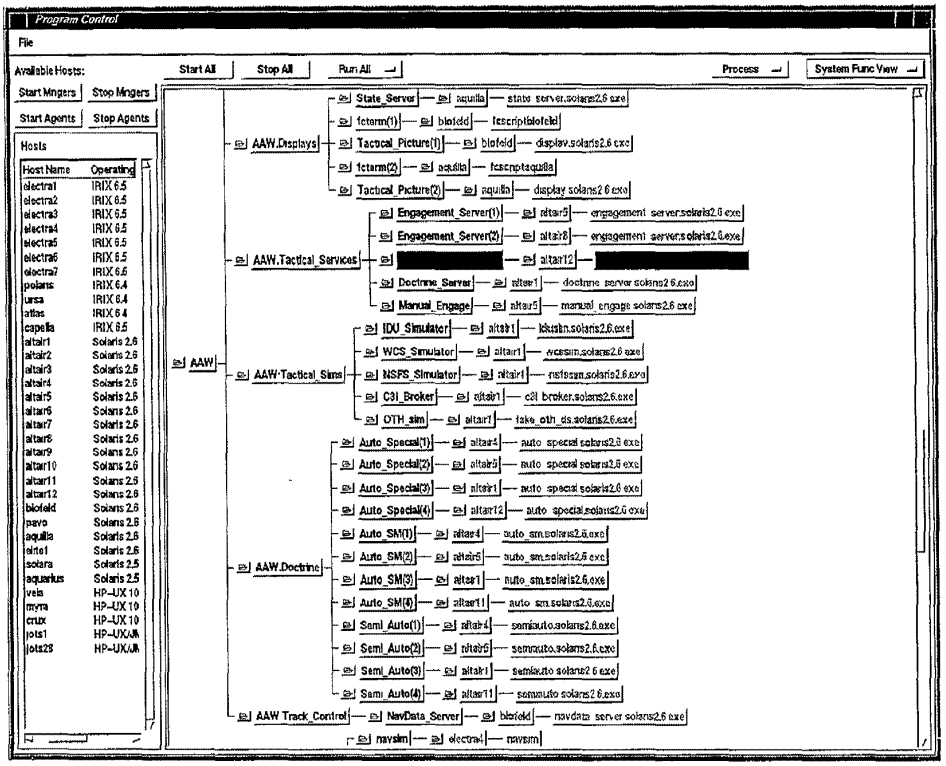


Figure 3.4.3.1-2. Program Control System-Level View.

f. Upon establishing a "configuration" and starting the infrastructure components, the operator has two options to point and click the applications and systems they wish to start from any of the three views. First, there is a manual mode that allows the operator to select the set of applications they wish to start and then initiate the request. In addition, there is a one-button startup capability that allows an operator to start a pre-defined default set of applications.

g. Since not all control functions are performed by the operator, a socket-level interface is provided to the display for the resource-manager to connect to and issue startup, shutdown, and allocation commands. This feature relies heavily on the System Specifications files. These files provide all the information the resource-manager needs to allocate, start, restart, and stop applications.

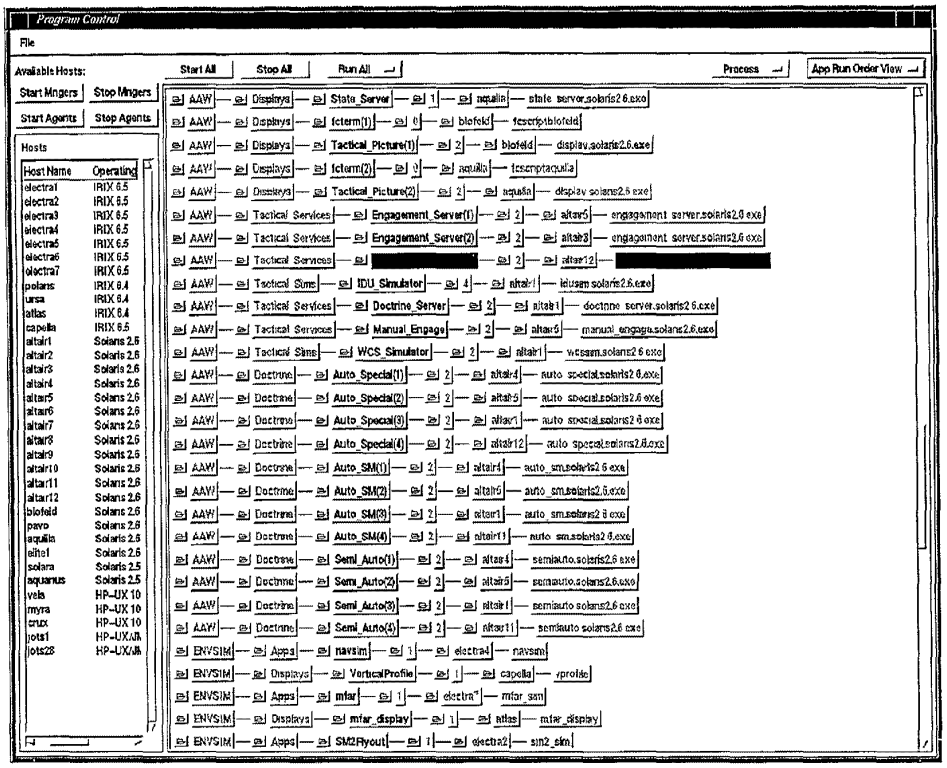


Figure 3.4.3.1-3. Program Control Run-Order View.

h. One important Program Control feature necessary for the resource-manager to dynamically start applications is the support for dynamic arguments in application command-line argument lists. Note the AutoSpecial command-line arguments from the configuration file example used earlier: "%(UNIQUE, 1, 32) A_Spcl_%(UNIQUE, 1, 32) -jewel -rstat –splot". This argument list requires AEC components to pass in a unique node number. In order for Program Control to know this on startup, we use the UNIQUE dynamic argument to generate a unique number between 1 and 32 within the AEC system. Other dynamic arguments include inserting date, time, and hostnames for other applications already running. The latter is important for client-server architectures that do not use location-independent mechanisms for the location of the server. Therefore, clients can use dynamic arguments to find out where their server is located.

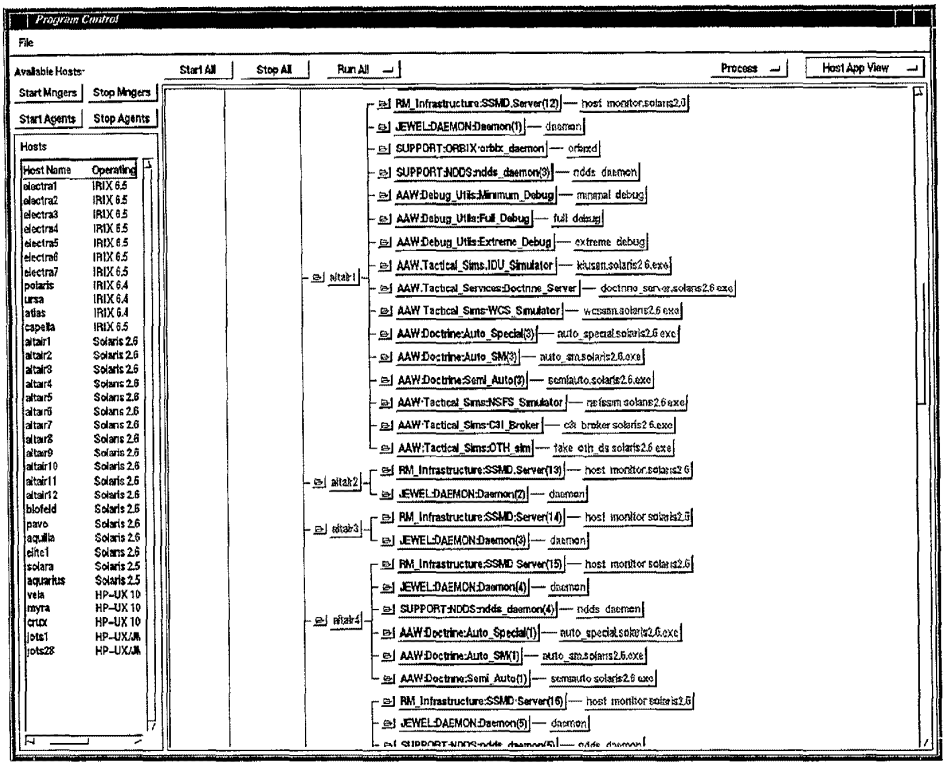


Figure 3.4.3.1-4. Program Control Host View.

3.4.3.2 Subsystem Managers.

In order to maintain a tiered architecture, the graphical-interface communicates with a set of managers. The purpose of each manager is to start its corresponding agents. Each manager is configured to handle up to ten agents. Currently they act as a "pass-through" in order to reduce the number of open sockets seen by the display process, yet capable of performing more functionality in future development.

3.4.3.3 Host Agents.

a. The Program Control agents are daemon servers that reside one per host. Their main function is to provide a socket interface allowing remote startup and shutdown of various application binaries. In addition, they provide feedback as to the up/down status of the processes they are controlling.

b. Demo98 support was restricted to the UNIX platforms Sun, SGI, and HP. This solution used the "fork" and "exec" system calls to create new application processes and captured UNIX SIGCHLD signals to detect application shutdown.

c. Special care must be taken when capturing asynchronous signals (SIGCHLD) and updating static tables. Race conditions are very likely to occur such that when the process

returns from a signal handler, recovery is not transparent. The best solution to this problem was to queue process shutdowns and handle them in a synchronous loop.

3.4.3.4 Summary.

Figure 3.4.3.4-1 represents the architecture with all three components. IP Multicast was selected as the form of socket communications. It proved to be fast and efficient, but occasionally unreliable and tricky to fragment packets by the sender and rebuilt by the receiver. Reliability was improved by increasing the socket send and receive buffers up to 256k bytes. This allowed processes to get further behind and still have free buffer space for newly arriving packets, or packets waiting to be sent over the wire. Future development will explore a TCP/IP solution.

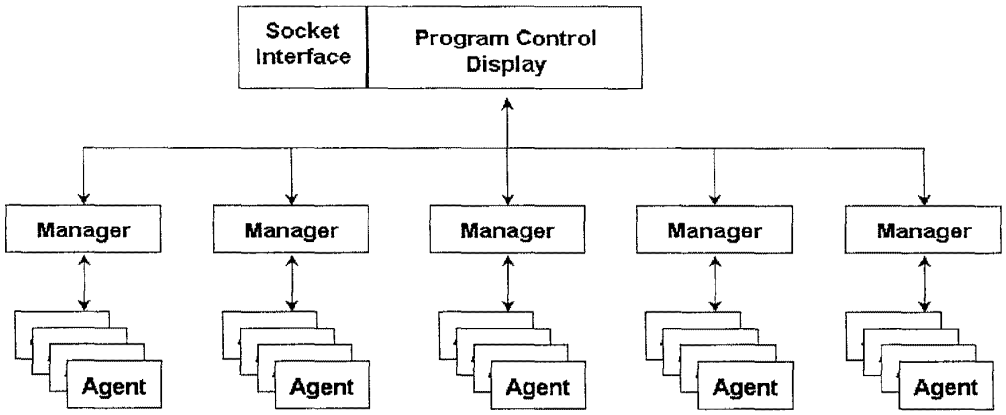


Figure 3.4.3.4-1. Program Control Architecture.

3.4.4 QoS and System Specifications.

To effectively manage a pool of computing resources, the Resource Manager must have some means of determining the capabilities and configuration of the computing resources under its control, of determining the software components that need to be executed and the dependencies of these software components on both hardware and software resources, determining what mission-level and application-level requirements are expected to be met, and determining what control capabilities are available to be used to attempt to recover from fault or QoS violation conditions. To address these needs, a System and Software Specification Grammar has been developed to attempt to capture the "static" information needed by the Resource Manager for effectively managing a pool of distributed resources. See Appendix G for details of the Grammar.

3.4.5 Visualization.

For Demo98, several new displays were developed and/or enhanced to showcase new Resource Management capabilities. The main display efforts for Demo98 consisted of enhancements to the Host Display, the Graph Display, the Path Display, and the development of the Resource Management Decision Review Display. Each of these display efforts is described below. (Several enhancements were also made to the Jewel Instrumentation Displays, in particular the Multi-AutoSpecial Doctrine display. These enhancements will not be discussed in this section since they have been described in detail in other sections of the report.)

3.4.5.1 Host Display.

The Resource Management host display depicts the layout of all machines that are stationed in either testbed and shows the processes that are running on each of the hosts along with each host's connections to the three networks: ATM, Ethernet, and FDDI. The information from which this picture is constructed is received from the data server using the ATDNET communications package. The data server collects information from the host monitor and passes the information along to the host display. Figure 3.4.5.1-1 is an example of the Host Display from the SCL. Another function of the host display is to allow the user to choose up to twelve hosts and submit requests to the data server for information collected from host monitor such as CPU usage, memory usage, packets in, packets out, and paging information. These parameters are displayed with respect to current time as line graphs by the graph display. Figure 3.4.5.1-2 is an example of a graph display from the SCL. The code for the host and graph displays was written in the C language. This code utilizes OpenGL libraries for drawing graphics and Motif libraries for user interface.

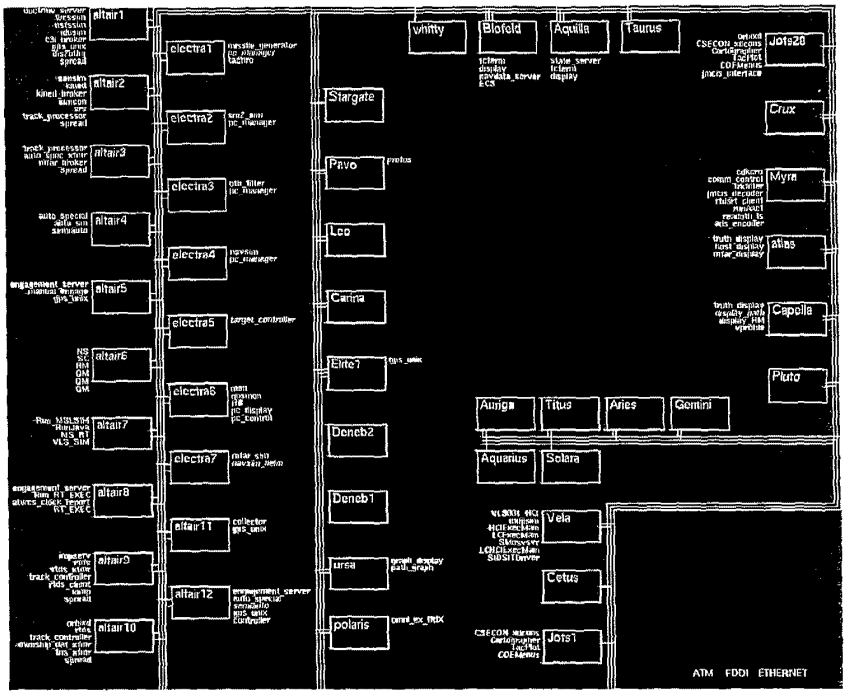


Figure 3.4.5.1-1 Host Display

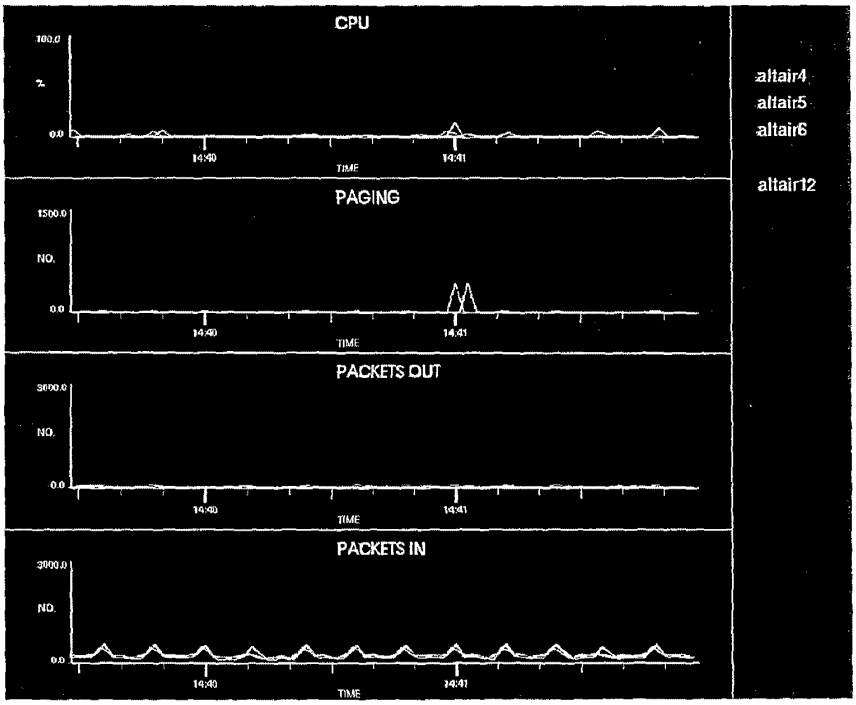


Figure 3.4.5.1-2 Graph Display

3.4.5.1.1 Host Display Design.

The host display initially reads an ASCII text file that contains information on all the hosts that are situated in the testbed. A separate file is created for each location listing information on each host such as the host name, host platform, host operating system, and statuses of host's network connections. Using the information in the file, the display shows a layout of all the hosts with boxes representing the hosts and color-coded lines illustrating the possible connections of each host to the three networks. This display is then updated on a continuous basis, based on messages received periodically from the data server. During this update process, box color is used to indicate the current status of each host and each color line drawn to a box indicates the host's specific network connection. There may be up to six processes listed next to each box denoting the processes being executed on the host. The process name color is used to indicate the status of each process (i.e. initializing, running, or faulted). As the display is running, the user may request to see actual system information for each of the hosts. This is accomplished by selecting up to twelve hosts and may be executed by clicking the left mouse button on each host box. After all the required hosts are selected, the middle mouse button is pressed to bring up a dialog box showing five different toggle buttons for selection of CPU, memory, packets in, packets out, and paging information. As each button is toggled, a request is made to both the data server for the specified information and the graph display for graphing of such information. The graph display may show up to four line graphs (arranged vertically). Untoggling a button allows the host display to send a stop data request to the data server and the graph display disabling both the transmission and graphing of data.

3.4.5.1.2 Data Formats.

This section illustrates the configuration information that the host display utilizes in order to render a layout view of all the hosts, their respective network connections, and the processes that are being executed on them. The information comes initially from a configuration file and is then periodically updated from the data server.

3.4.5.1.2.1 Host Configuration File.

a. To initially depict the layout of the hosts, the host display must read in a host configuration file and keep track of the information in a database based on the host name. The host display updates its database with the host information. Each host box initially comes up gray with only the host name in the box, and no network connections is displayed. An example of the configuration file is shown in Figure 3.4.5.1.2.1-1.

Host Configuration File
Taurus ALPHA_200 DIGITAL_UX 180.0 1 0 1 Aquilla SPARC_2 SunOS_5.6 180.0 1 1 1 Leo ALPHA_200 DIGITAL_UX 90.0 1 0 1 Stargate MICRON WINDOWS_95 180.0 1 0 0 Carina SPARC_1 SunOS_5.6 90.0 1 0 1 Pluto INDIGO IRIX_5.3 270.0 1 0 1 Myra HP_J210_9000 HP-UX_B.10.20 270.0 1 0 1

```
Electra1 ORIGIN_200 IRIX64_6.4 90.0 1 1 0
Altair1 ENTERPRISE_2 SunOS_5.6 270.0 1 1 0
Whitty PENTIUM_90 WINDOWS_95 90.0 1 0 0
```

Notes:

1st parameter – host name
2nd parameter – host platform [default]
3rd parameter – host operating system [default]
4th parameter – network connection side (0.0 - top, 90.0 - left, 180.0 – bottom, 270.0 - right)
5th parameter – ethernet network status (0 - down, 1 - up) [not used]
6th parameter – atm network status (0 - down, 1 - up) [not used]
7th parameter – fddi network status (0 - down, 1 - up) [not used]

Figure 3.4.5.1.2.1-1 Host Configuration File

b. Each line in the configuration file contains information pertained to a host. The first three parameters identify the host's name, platform, and operating system respectively. The fourth parameter indicates the side of the box where the network lines are to be drawn. A number 0.0 indicates lines will be drawn from the top of the box, 90.0 means from the right of the box, 180.0 means from the bottom of the box, and 270.0 indicates from left of the box. The last three parameters show the statuses of the Ethernet, ATM, and fddi network connections respectively.

c. The host display allows up to fifty hosts to be shown so the configuration file may contain up to fifty hosts. The locations of all fifty boxes have already been configured and saved to a default config file, but the user may change these locations by invoking the config option in the display and providing new location values. The user can toggle the "i" key in the display to show the numbering of host boxes corresponding to the ordering of the hosts in the configuration file. This process may be helpful in creating a host configuration file.

3.4.5.1.2.2 Interface to Data Server.

There are two messages that are received from the data server. These consist of the host config message and the host process message. The host config message provides the name, status, platform, and operating system of each host. The host process message provides the three network statuses of each host and the status of any processes that are running on that host.

3.4.5.1.2.2.1 Host Configuration Message.

a. Figure 3.4.5.1.2.2.1-1 shows the structure that the host display uses to indicate whether the host is up or not by box color and to show the host name, platform, and operating system in the box.

Host Configuration Message

```
Typedef struct
{
   char hostname[GRAPH_HOSTNAMELEN];
   int status;                          /* Host status (0 = down, 1 = up) */
   char hosttype[GRAPH_HOST_TYPE_LEN];  /* Type of machine */
   char os_type[GRAPH_OS_TYPE_LEN];     /* Operating system and version */
} graph_host_config_message;
```

Figure 3.4.5.1.2.2.1-1 Host Configuration Message

b. As the host display receives the message, it checks to see whether the host is in its database. If the host is in the database, then the host display updates the database with the information and changes the display accordingly. If the host is up and running, the display changes the box color to blue and shows the host name, platform, and operating system. If the host is not responding to monitoring, then the box color is gray and shows only the name of the host in the box.

3.4.5.1.2.2.2 Host Process Message.

a. Figure 3.4.5.1.2.2.2-1 shows the structure of the host process message that the host display uses to illustrate network connections along with the any processes that are being executed on all the hosts.

Host Process Message

```
Typedef struct
{
   char process_name[GRAPH_PROCESS_NAME_LEN];
   unsigned pid;                    /* Process ID */
   int status;                      /* Process status */
                                    /* 2 – process is new */
                                    /* 1 – process is running */
                                    /* -2 – process just died */
                                    /* -1 – process is dead */
   unsigned long int mem_size;
   float cpu_time;
} graph_process_data;

typedef struct
{
   char hostname[GRAPH_HOSTNAMELEN];
   int status;                      /* Host status (0 = down, 1 = up) */
   int ether_status;                /* Ethernet connected (0 = down, 1 = up) */
   int fddi_status;                 /* FDDI connected (0 = down, 1 = up) */
   int atm_status;                  /* ATM connected (0 = down, 1 = up) */
   double timetag;
   int num_processes;
```

```
  graph_process_data processes[GRAPH_NUM_PROCESSES];
} graph_host_process_message_format;
```

Figure 3.4.6.1.2.2.2-1 Host Process Message

b. As the host display receives the message, it checks to see whether the host is in the database. If the host is in the database, then the host display updates its database and makes changes to the display accordingly. The host box color changes according to the host's status, and network lines are drawn to connect to the host box if the statuses of the networks are up. For the host's processes, a yellow color process name indicates that process is new, a green color indicates that process is running, and a red color means that process has faulted.

3.4.5.1.3 Graph Display Interface.

a. The host display has an interface to the graph display to enable graphing of CPU usage, memory usage, packets in, packets out, and paging information. Once the graphs are enabled by user input from the host display, the graph display shows the information received from the data server as line graphs with respect to current time. The graph display keeps track of the information in databases. Up to four graphs may be displayed at any time, and within each graph, there's a limit of up to twelve host lines presenting the actual system performances. Each of these lines is drawn from a database containing info on up to 500 points.

b. The graph display has a user interface that allows the user to change such configuration items as y scaling of each graph and time length shown on each graph.

3.4.5.1.4 User Interface.

The user may change colors, fonts, and configurations of the components of the display by selecting color button, font button, or config button in the popup menu. To enable/disable graphing of CPU usage, memory usage, packets in, packets out, and paging information in the

graph display, a dialog box may be invoked to allow toggling of the five buttons. The following shows the effects of entering key inputs:

"d" - toggle debug mode
"i" - toggle show numbering of boxes
"n" - toggle show network connections
"p" - toggle show processes
"r" - clear the toggle buttons of graph requests
"t" - save process names to file
"u" - toggle show last update time of host box
"w" - toggle show time when display is updated
"ESC" – exit

3.4.5.2 Path Display

The Resource Management path display shows the flow of data and scaling of processes for five different paths: Auto-Special Periodic Review Path, Spy Declared Auto Special Path, Semi-Auto Periodic Review Path, Auto-SM Doctrine Path, and ATWCS Path. The purpose of this display is to demonstrate the distribution of data and its effect on process scaling for each of these paths as track load is varied. Using the ATDNET communications package, the path display receives the information from the data server, which collects the data from the host monitor. The path display also shows application performances of certain processes within each path. This information is received from the QoS monitor. The code for the path display was written in the C language using OpenGL libraries for drawing graphics and Motif libraries for implementing user interface.

3.4.5.2.1 Path Display Design.

The information to construct the path display is received from two interfaces, the data server and the QoS monitor. The upper half section of the display shows information obtained from the data server. It consists of process names in the order that forms the data flow for a path and boxes underneath each process name representing the number of copies (up to six) of process running on all the host machines being monitored. The color of the box is used to indicate the status of the process (i.e. running and faulted), and the name on the box is the host on which the process is being executed. Arrows are drawn from the left to right direction of the sequence of boxes to indicate the data flow of the processes. At the lower half section of the path display, application performances obtained from the QoS monitor is presented as line graphs with respect to current time. Up to two graphs may be invoked by the user with key inputs or selecting the time lines underneath the specific process.

3.4.5.2.2 Data Display

This section deals with two types of information being shown on the path display. The first pertains to data flow of process scaling, and the second includes application performances.

3.4.5.2.2.1 Data Flow

a. There are twelve processes that comprise the five different paths: Auto-Special Periodic Review Path, Spy Declared Auto Special Path, Semi-Auto Periodic Review Path, Auto-SM Doctrine Path, and ATWCS Path. These processes are as follows: track processor, rtds, auto-special, engagement server, wcs_sim, track control, semi-auto, auto-sm, nav_sim, navdata server, lc_rt, and vls_sim. Figure 3.4.5.2.2.1-1 shows the data flow for all the five paths.

AUTO-SPECIAL PERIODIC REVIEW PATH
TRACK PROCESSOR → RTDS → AUTO-SPECIAL → ENGAGEMENT SERVER → WCS_SIM
SPY DECLARED AUTO SPECIAL PATH
TRACK CONTROL → AUTO-SPECIAL → ENGAGEMENT SERVER → WCS_SIM
SEMI-AUTO PERIODIC REVIEW PATH
TRACK PROCESSOR → RTDS → SEMI-AUTO → ENGAGEMENT SERVER → WCS_SIM
AUTO-SM DOCTRINE PATH
TRACK PROCESSOR → RTDS → AUTO-SM → ENGAGEMENT SERVER → WCS_SIM
ATWCS PATH
NAV_SIM → NAVDATA SERVER → LC_RT → VLS_SIM

Figure 3.4.5.2.2.1-1 System Path Data Flow

b. As the data server reports the number of copies for each process observed to be running on host machines, the path display updates the view by drawing the corresponding number of boxes with the host names in the boxes underneath the process name. Box color is blue when the process is observed to be running, and box color is gray when process has faulted. The box goes away completely when the data server notifies the death of a process. Figure 3.4.5.2.2.1-2 shows an example of a Semi-Auto Periodic Review Path display.

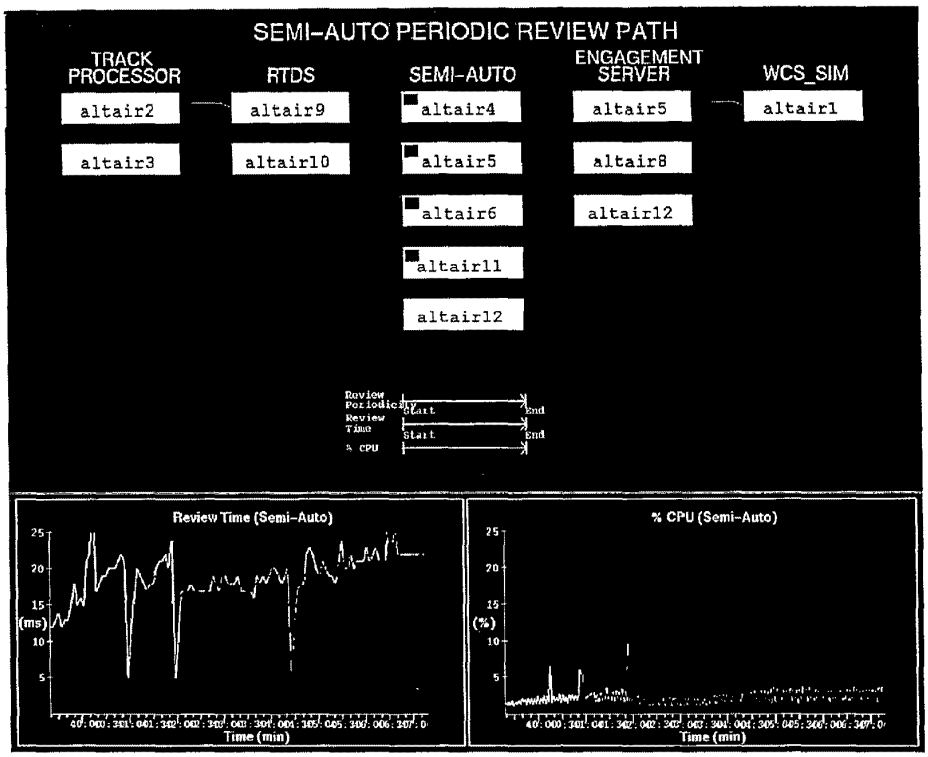


Figure 3.4.5.2.2.1-2 Path Display

3.4.5.2.2.2 Application and Path Performance Data

a. Figure 3.4.5.2.2.2-1 shows the available application and path performance data for each path. The figure also shows the application component(s) from which the data is being measured.

AUTO-SPECIAL PERIODIC REVIEW PATH	
PROCESSES	**PERFORMANCES**
AUTO-SPECIAL	**Track Load** **Review Time** **% CPU** **Review Periodicity**
SPY DECLARED AUTO SPECIAL PATH	
PROCESSES	**PERFORMANCES**
TRACK CONTROL **AUTO-SPECIAL** **ENGAGEMENT SERVER**	**Review Time** **% CPU**
TRACK CONTROL → ENGAGEMENT SERVER	**Path Review Time** % CPU
SEMI-AUTO PERIODIC REVIEW PATH	

PROCESSES	PERFORMANCES
SEMI-AUTO	**Review Periodicity** **Review Time** % CPU
AUTO-SM DOCTRINE PATH	
PROCESSES	PERFORMANCES
AUTO-SM	% Late Tracks % CPU
ATWCS PATH	
PROCESSES	PERFORMANCES
NAV_SIM → VLS_SIM	Nav Data Delivery Latency Missile Alignment Latency
LC_RT	% CPU

Figure 3.4.5.2.2.2-1 Path Performance Data

b. The user may submit requests to the QoS monitor to see specific application performances by key inputs or by selecting the time lines underneath the processes. From the information that is received from the QoS monitor, the path display keeps these parameters in databases in order to update the line graphs.

3.4.5.2.3 User Interface.

The path display provides a user interface to allow some flexibility in displaying the view. The ability to change the colors, fonts, and configurations of various components of the display is accomplished by selecting the color button, font button, or config button in the popup menu. Specific to the path display are the following user interface capabilities: selection of each view of the five paths by selecting the corresponding button in the popup menu, selection of graphs by selecting the time lines drawn specifically for the processes, and key inputs for specific events. The following are the effects of entering these keys:

"1" - show AUTO-SPECIAL PERIOD REVIEW PATH
"2" - show SPY DECLARED AUTO SPECIAL PATH
"3" - show SEMI-AUTO PERIODIC REVIEW PATH
"4" - show AUTO-SM DOCTRINE PATH
"5" - show ATWCS PATH
"a" - show first graph of path
"b" - show second graph of path
"c" - show third graph of path
"d" - show fourth graph of path
"e" - show fifth graph of path
"f" - show sixth graph of path
"g" - show seventh graph of path
"h" - show eighth graph of path
"0" - set the y scale of the displayed graphs to default y scale value
"8" - set the y scale of the displayed graphs to 4 times the default y scale value

"9" - set the y scale of the displayed graphs to 2 times the default y scale value
"+" - increase time length of the graphs by 1 minute
"-" - decrease time length of the graphs by 1 minute
"ESC" – exit

3.4.5.3 Resource Management Decision Review Display.

The Resource Management Decision Review display (Figure 3.4.5.3-1) shows Resource Management control events such as selection of where to start an application, recovery of a faulted application, and scaleup of an application in response to overload conditions. A scroll list displays a history of the Resource Management events. Host load information for the five best potential hosts for where to place an application to recover from a fault or overload condition is displayed as a set of bar graphs. A text display of the last three events is also shown. The information displayed is received from the Resource Manager. The Decision Review Display is written in C using OpenGL libraries for the graphics primitives and Motif libraries to implement the user interface for the display.

3.4.5.3.1 Design.

The layout of the RM justification display is assembled into three different sections. The first (top left) section shows a list of the events history (the last 256 events) divided into five columns: Event #, Action Type, Application Name, Host Event, and Host Action. The list has scrollbars along the right and bottom edges that allow vertical and horizontal scrolling, and the portion of the list data that is visible pertains to the last few events. Also, single selection of the events in the list is supported to allow the second section to display information pertaining to the selected event. The default selection of an event is the current event. The second (bottom left) section shows four bar graphs displaying such information as aggregate score, CPU score, network score, and memory score for up to five hosts as each event is received from the resource manager. The second section may also show a line graph depicting scale up information if an event selected from the list is an action type of application scale up. This graph may be invoked by toggling the "Scale-Up Plot" button at the top of the history of events list. The button is sensitized for toggling only if the selected event in the list contains the scale up application action. The third (right) section shows a text display of the last three events. The order of the three events consists of the current event being placed at the top, the previous event in the middle, and the event previous to the middle event at the bottom of the section.

Figure 3.4.5.3-1. Resource Management Decision Review Display

3.4.5.3.2 Data Formats.

This section introduces the two messages received from the resource manager to depict the RM justification display.

3.4.5.3.2.1 Event Message.

a. Figure 3.4.5.3.2.1-1 shows the structure of the event message that the RM justification display utilizes to draw all three sections of the display.

EVENT MESSAGE

```
typedef struct
{
  char event_app_name[MAX_CHARS];     /* application name associated w/ event */
  int event_pid;                      /* UNIX pid of event */
  char event_host_name[MAX_CHARS];    /* host on which event occurred */
  double event_time;                  /* time event occurred in seconds */
  int event_num;                      /* event id number */
  int event_type;                     /* event type
                                         0 = request start of application
```

```
                                        1 = failure of application
                                        2 = application overload */
int action_type;                      /* action type
                                        0 = application started
                                        1 = application restarted
                                        2 = application scale up */
char action_host_name[MAX_CHARS];     /* host on which action occurred */
int action_pid;                       /* UNIX pid of action */
double action_time;                   /* time action occurred in seconds */
double response_time;                 /* time it took the RM to response
                                         defined as action_time – event_time */
double total_action_time;             /* time tag for Program Control's
                                         response to RM */
double total_response_time;           /* delta time between Program Control's
                                         response to RM and the event time */
char hostnames[5][MAX_CHARS];         /* array of 5 best host choices */
float agg_data[5];                    /* aggregate score data for 5 hosts */
float cpu_data[5];                    /* CPU score data for 5 hosts */
float net_data[5];                    /* network score data for 5 hosts */
float mem_data[5];                    /* memory score data for 5 hosts */
} event_message_type;
```

Figure 3.4.5.3.2.1-1. Event Message

b. The first section of the display lists the events as they are received showing the following information taken from the parameters of the event message structure: the event number, the action type (application started, application restarted, or application scale up), the application name, the name of the host on which the event occurred, and the name of the host on which the action occurred. The events are stored in a linked list of up to 256 events so only the last 256 events are displayed in the list. The ability for the user to select an event in the list is implemented to allow the second section to display information pertaining to the selected event. The default selection of the current event may be invoked by pushing the "Display Current" button from the popup menu.

c. The second section of the display uses the parameters (hostnames, agg_data, cpu_data, net_data, and mem_data) of the event message structure to display the four bar graphs showing aggregate score, CPU score, network score, and memory score of up to five hosts for the selected event. Information of the hosts may be displayed with different colors of the bars representing different hosts. As an event is selected in the list of the first section of the display, the bar graphs alter to show the information for the selected event.

d. The third section of the display uses the parameters of the message structure to fill in the information shown in Figure 3.4.5.3.2.1-2. The current event information is placed at the top of the section, the previous event is placed in the middle, and the event previous to the middle event is placed at the bottom of the section.

```
APPLICATION:                                EVENT #:

EVENT:
                PID:        ON HOST

EVENT TIME:

ACTION:
                PID:        ON HOST

ACTION TIME:

RESPONSE TIME:
```

Figure 3.4.5.3.2.1-2. Event Text Display

3.4.5.3.2.2 Scaleup Message.

a. Figure 3.4.5.3.2.2-1 represents the scale up message that the RM justification display receives from the resource manager.

SCALE UP MESSAGE

```
typedef struct
{
  double timetag;
  float value;
} event_plot_type;

typedef struct
{
  int event_num;
  double start_time;
  double stop_time;
  double event_time;
  float min_value;
  float max_value;
  float threshold;
  char threshold_string[MAX_CHARS];
  char axis_legend[MAX_CHARS];
  char title[MAX_CHARS];
  int num_data_points;
  event_plot_type event_data[MAX_DATA_POINTS];
} scale_up_message_type;
```

Figure 3.4.5.3.2.2-1. Scaleup Message

b. This information is displayed in the second section of the display if the "Scale-Up Plot" button is toggled. The second section consists of texts showing the event time, start time, stop time, minimum value, maximum value, and threshold value of the scale up action event.

The second section also shows a line graph of the scale-up values for the time tags along with a line at the threshold value running across from the start time to the stop time.

3.4.5.3.3 User Interface.

The RM justification display implements a user interface to allow changes to the colors, fonts, and configurations of the components in the display. The user interface also allows the user to clear the events shown on the display by pushing the "Reset Button" button in the popup menu. Also, to set the current event to be selected, the user pushes the "Display Current" button in the popup menu. Another user interface of the RM justification display is the "Scale-Up Plot" toggle button allowing the user to see the scale up information of an application scaleup action event.

3.5 Demo 98 Hardware Configuration

All the hardware used in Demo 98 was COTS equipment. The hardware configuration and computer program allocation are shown in Figure 3.5-1. This is a very heterogeneous system composed of equipment from four vendors, using four operating systems and three networks.

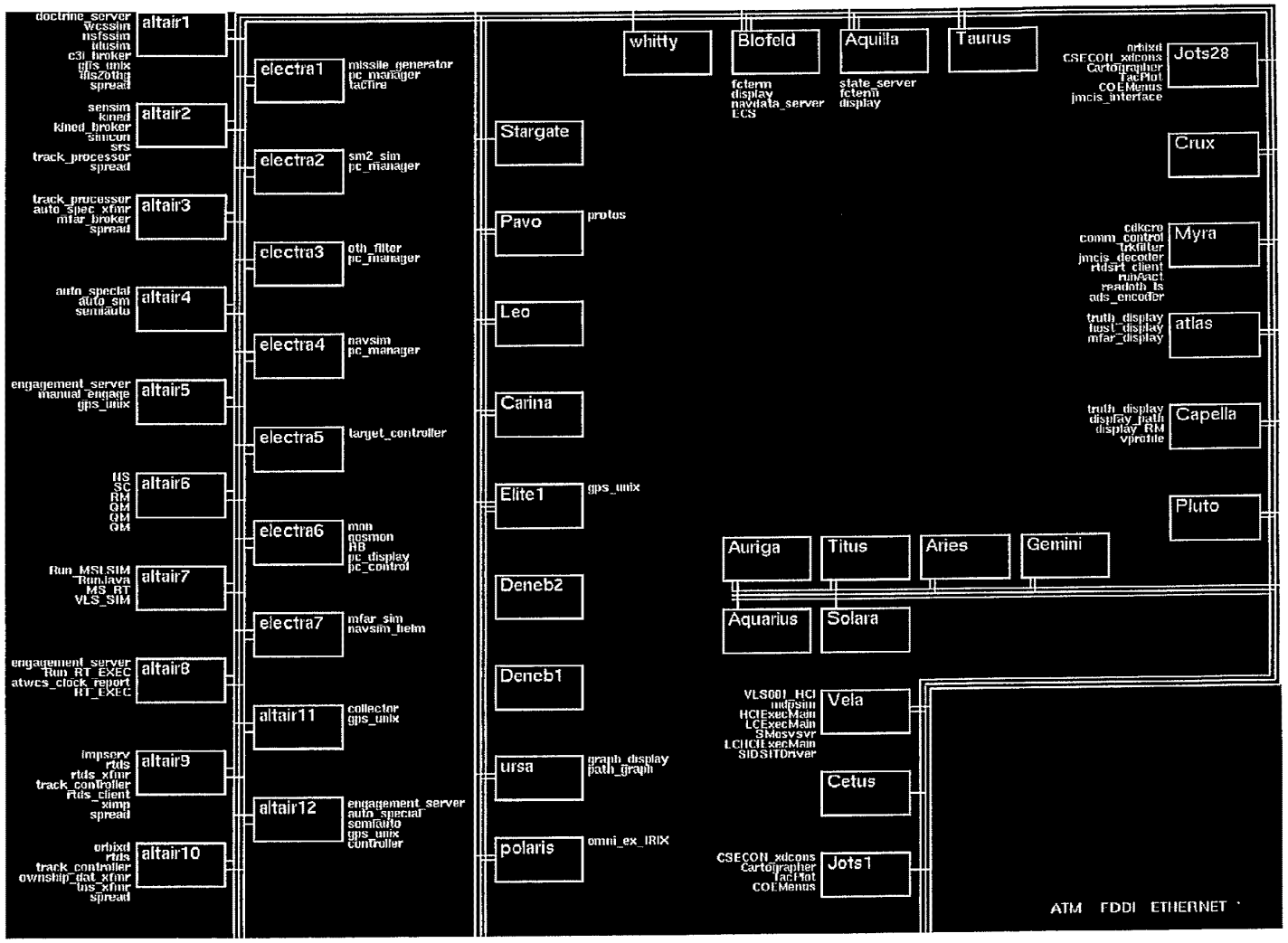


Figure 3.5-1 Demo 98 Hardware and Software Configuration

3.6 Demo 98 Scenario

a. The scenario for Demo 98 was developed based on the Surface Combatant-21st Century (SC-21) Cost and Operational Effectiveness Analysis (COEA) scenario. A description of the geographic setting and tactical objectives follows.

b. In 2015, the Korean Peninsula is invaded. In one of the responses, a surface battle group consisting of ownship, one CG and two DDGs is dispatched as shown in Figure 3.6-1.

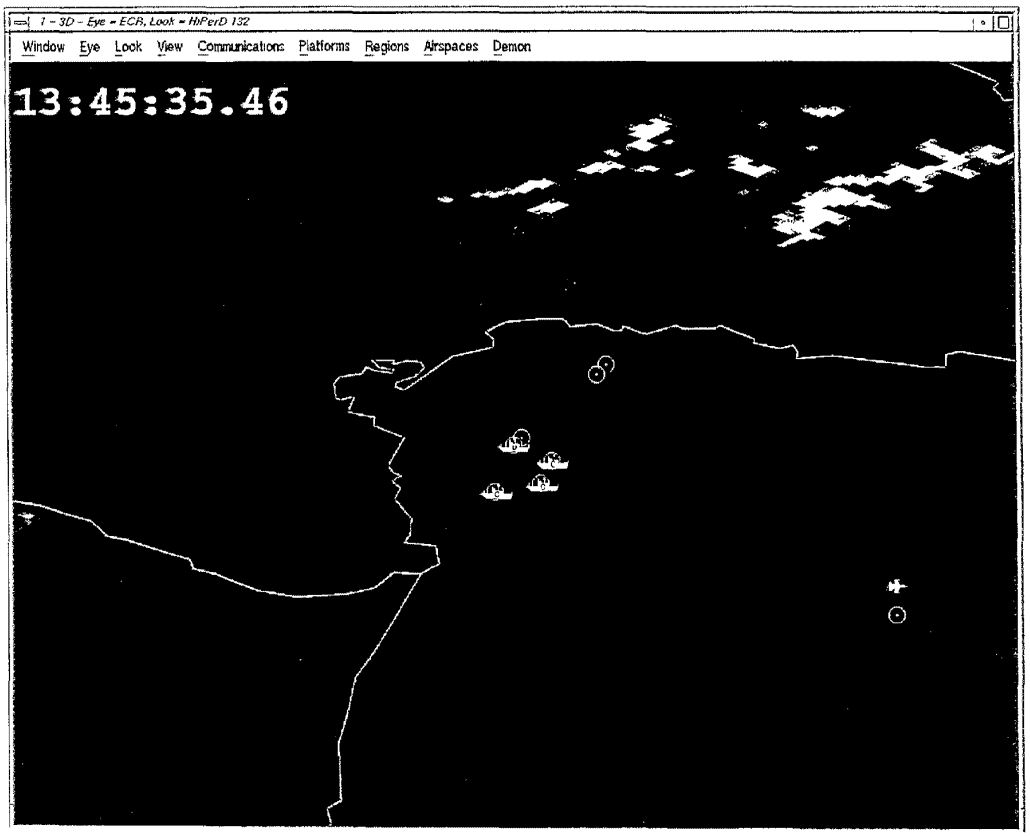


Figure 3.6-1. Surface Battle Group

c. The battle group's mission is two-part: to gain sea battlespace dominance, and to support land attack. Ownship mission is to coordinate Naval surface fire within the battlegroup. This will entail strategic attack and invasion slowing missions accomplished with the Advanced Tomahawk Weapon Control System (ATWCS); scheduled and unscheduled digital calls for Naval Surface Fire Support (NSFS); AAW self defense; and air space deconfliction.

3.7 Integrated System Demonstration

The Integrated System Demonstration began with a demonstration of the new physics based DIS compatible Environmental Simulation capability. This was followed by two ATWCS land attack strike missions with the second mission including a fault detection and recovery sequence. Next the Fault Tolerant Engagement Server was presented and the impact of its faulting during an Aegis SPY Auto-Special engagement was shown. Then a Digital Call for Fire sequence was demonstrated. Finally the system track capacity was scaled up as Resource Management functionality was demonstrated. It was shown that some of the RM components are themselves fault tolerant, that RM recognizes components not meeting their specification as track load increases, and reallocating to allow the system to regain expected performance, and that RM recognizes software faults and reallocates the faulted process to a different processor. At the end of the demo with a track load of approximately 7500, a series of Auto-Special engagements were run showing that the system still met critical timeline requirements even under very heavy loads. The following subparagraphs describe these demo events in detail.

3.7.1 Environmental Simulation

a. The integrated demonstration began with an Environment Simulation scenario that entered ownship, three ships in company, an AEW aircraft flying a race track pattern, and eight land sites detected by the AEW aircraft. These tracks are shown in the OTH Filter display in Figure 3.7.1-1. This portion of the demonstration was used to introduce the various Environment Simulation displays such as the Truth displays, the Helm Control display, the Vertical Profile display, the MFAR display, and the OTH Filter display. Also the AAW Tactical display was compared with the JMCIS display to illustrate that a consistent track picture was provided by both. The AAW and OTH tracks were received and displayed by both systems.

b. A second EnvSim scenario depicts an enemy aircraft attack against ownship. As depicted in Figure 3.7.1-2, an enemy aircraft starts 37 nmi from Ownship at an altitude of 50'. At this point it is detected by the OTH sensor (the AEW aircraft), but not by the ownship radar (MFAR) as the enemy aircraft is below the MFAR radar horizon. It is displayed on both the AAW Tactical Picture display and the JMCIS C4I display as an OTH track, the track being provided by the OTH sensor to JMCIS and then onto the AAW Tactical Picture. As it moves on, the enemy aircraft pops up to 2000' at 33 nmi for 35 seconds to target ownship. When it pops up it is detected by the ownship MFAR and displayed on the AAW Tactical Picture as an AAW track with the AAW track information passed on to JMCIS. On the AAW Tactical Picture display the OTH track symbol is replaced with an AAW track symbol.

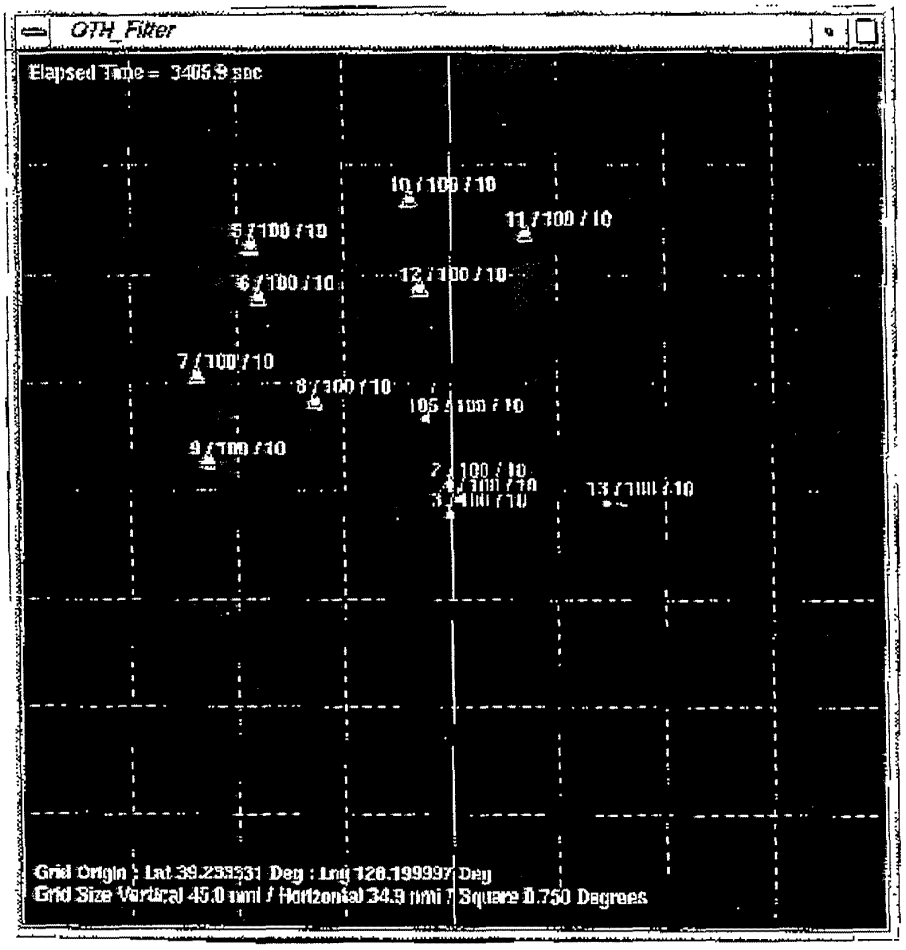


Figure 3.7.1-1. OTH Filter Display

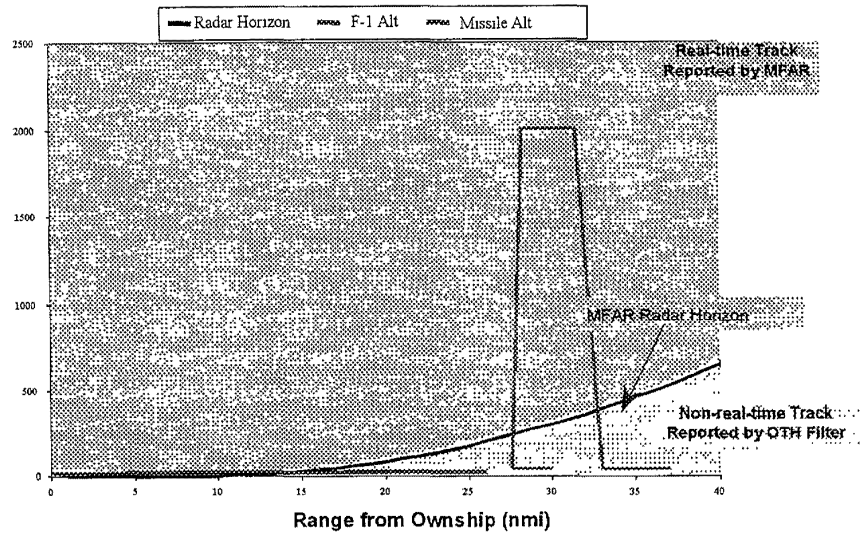


Figure 3.7.1-2. ASM Launch Scenario

c. The enemy aircraft then descends back to 50' (below MFAR horizon) and launches two ASMs against ownship. Once the AAW track is lost by MFAR the AAW track symbol on the Tactical Picture display is replaced with the OTH track symbol as the OTH sensor is still reporting the track. The two ASMs launched at ownship appear at first as OTH tracks on the AAW Tactical Picture as they are initially below the MFAR horizon. At 16 nmi from ownship the MFAR detects the two ASMs as SPY Auto-Special tracks, the OTH symbols are replaced with AAW track symbols on the Tactical Picture, and ownship using the auto-special doctrine automatically launches a SM-2 missile against each incoming threat. The environment simulation vertical profile display shows this attack sequence and the ownship response.

d. This scenario segment illustrated the OTH track input into the system via JMCIS and the AAW track being entered into the system via the physics-based MFAR simulation. It also demonstrated how the OTH and AAW tracks were shown on the Tactical Picture during the phases of the scenario segment. The Vertical Profile display was used to show the attack sequence and thus allowing correlation between what the scenario was doing and what the AAW Tactical Picture and JMCIS were showing. The defense of ownship via the Auto-Special doctrine was also illustrated.

e. In the third EnvSim scenario, the CVBG (east of ownship) launches several strike forces and an Air Force strike force approaches from the south of ownship. This is shown on the truth display in Figure 3.7.1-3. This scenario continued for the length of the demonstration providing background tracks for system processing. The total number of tracks being processed and entered into the system by MFAR reached over 200 during the demonstration This was done to illustrate that Environmental Simulation could generate and process a realistic capacity of tracks in its initial entry into the testbed. Larger capacities could have been demonstrated and are planned for future demonstrations.

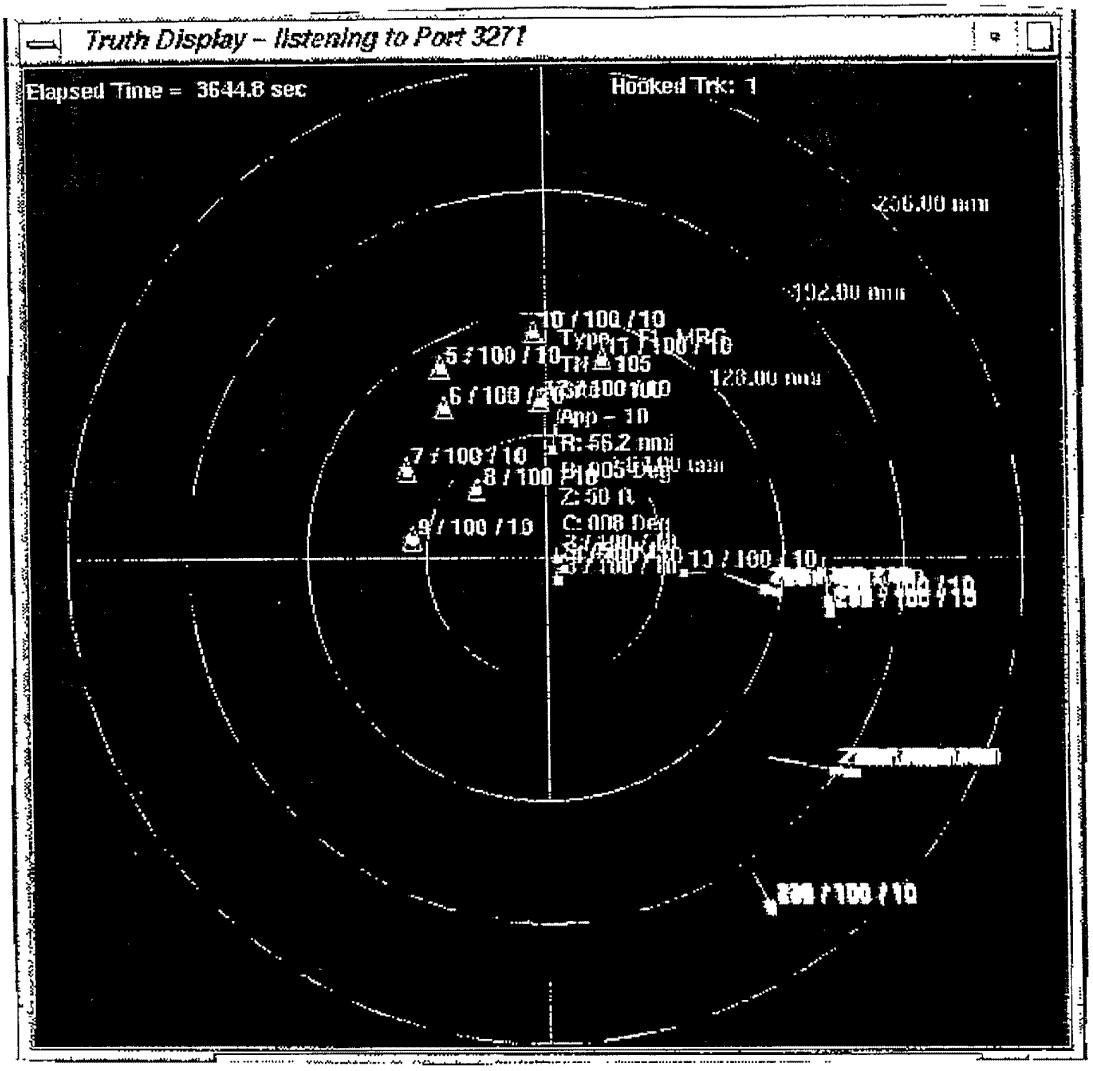


Figure 3.7.1-3 Truth Display Showing Air Targets

3.7.2 ATWCS Launch Control Real Time Group

a. The next segment of the demonstration was ATWCS which involved launching Tomahawk Land Attack Missiles (TLAMs) at predefined targets significant distances inland. The Tomahawk weapon system does not "target" hostile tracks, and does not rely on any real-time sensor system for targeting. (Real-time track reporting is essential for performing over-water routing by the launch platform. That part of the Tomahawk weapon system was not available for the demo.)" Currently, targeting is accomplished in advance of the launch, typically at a Cruise Missile Support Activity (CMSA) ashore. Using available imagery and intelligence data, the CMSA plans the overland route to the target from a point (the First Preplanned Waypoint, FPPWP) just prior to landfall. Included in the "mission" is the terminal attack profile, selected by the planner to maximize damage to the target.

b. The launch platform's responsibility is to:

(1) Meet any launch time and position requirements established by the tasking authority.

(2) Define the route from the launch point to the FPPWP.

(3) Initialize and launch the weapon.

c. The over water routing must consider the tactical surface track picture to avoid unintentional intrusion into "no-fly" zones, and to avoid collision with any hazards along the route.

d. To show the successful integration of the ATWCS LC capability into the demonstration, four engagement plans were created and executed. The first three plans specified the launch of two TLAMs each, with launch times set to require overlapping initialization of the six missiles. This allowed the demonstration to show that the testbed architecture and prototyped LC software successfully achieved the real-time performance required for TLAM launch. Figure 3.7.2 -1 shows the console display during missile alignment.

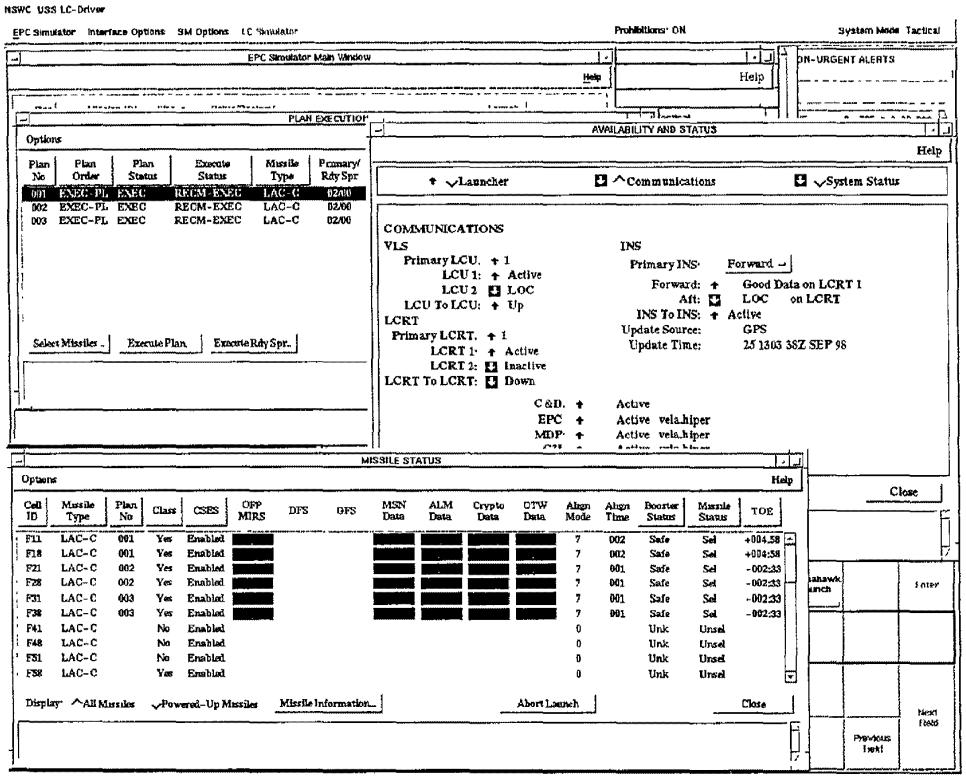


Figure 3.7.2-1. ATWCS Display Showing Missile Alignment

e. The fourth plan, calling for three missiles, was then executed to demonstrate the fault tolerance of the LC RT process, in conjunction with RM. During the missile initialization phase for this plan, the LC RT process was abnormally terminated to simulate a fault condition. This event was automatically detected by RM, which immediately issued a restart of the LC RT process. The restarted LC RT re-established communications with the LC Exec, the simulated INS, and the simulated VLS, then resumed the missile initialization process. As is seen in Figure 3.7.2-2 this resulted in the successful launch of the three TLAMs with a delay of a only few seconds.

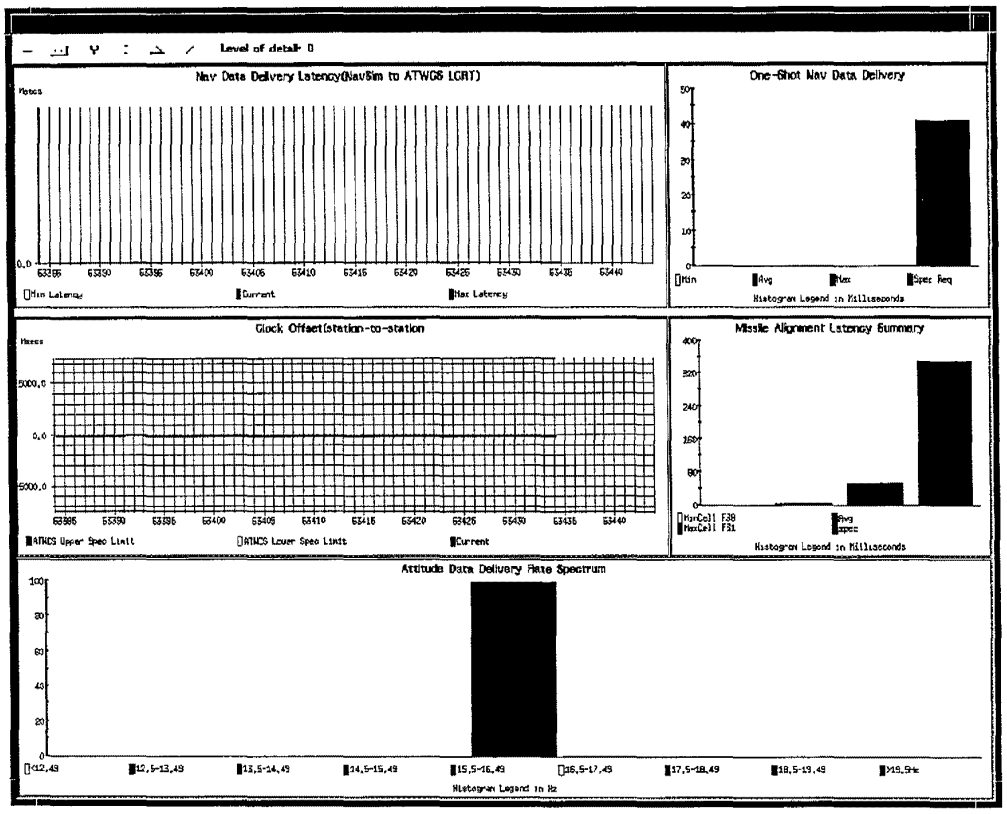


Figure 3.7.2-2 JEWEL Display Showing LC RT Fault and Re-start by HiPer-D Resource Manager

f. Had the LC RT not been restarted and communications with VLS resumed within an 80 second timeout period, VLS would have automatically deselected and safed the three TLAMs. This would have required the ATWCS operator to edit the engagement plan and restart missile initialization from the beginning, with a significant delay in launching the TLAMs.

3.7.3 Fault Tolerant Engagement Server

a. Prior to Demo 98 the Engagement Server component was not fault tolerant and, therefore, represented a single point of failure in the HiPer-D engagement capability. The purpose of the Engagement Server in HiPer-D is threefold. First, it validates engagement requests from clients and arbitrates any race conditions occurring due to multiple engagement requests on the same target. Second, it generates engagement orders to WCSSim for valid engagement requests. Third, it distributes engagement status updates to clients as tracks progress through their engagement sequence. In fulfilling these responsibilities the Engagement Server creates and maintains engagement state data critical to the HiPer-D system. It is imperative that this state data is synchronized among replicas when there is more than one Engagement Server executing in the system. The engagement servers use a Primary/Shadow scheme to enforce this synchronization and provide for fault tolerance.

b. During Demo 98 this new Fault Tolerant Engagement Server was demonstrated to properly handle engagements prior to, during, and after a fault of the primary replica. A fault resulting in a process failure was precisely injected into the primary Engagement Server during a SPY-declared Auto-Special engagement. This is the most demanding timing requirement in (C&D). The intentional failure was generated in the primary Engagement Server after validation checks had been performed but before the engagement order was sent to the WCSSim, (the worst possible time since it falls within the SPY-declared Auto-Special timing requirement). This failure location in the Engagement Server was chosen to demonstrate state data consistency among the surviving replicas as well as performance impact to the SPY-declared Auto-Special timeline.

c. To demonstrate the features of the Fault Tolerant Engagement Server a new JEWEL display was created which provides a visual window into the replicated Engagement Servers. This new display highlighted the consistency among the replicas with respect to their state as well as to the resultants of any input stimulus. Figure 3.7.3-1 shows an image of this display captured during one of the live Demo 98 executions. It was captured immediately after the replicas processed an engagement request for a SPY-declared Auto_Special contact.

d. There are three rows of colored bars on the display, one row for each of the Engagement Server replicas. The top row indicates the primary replica; the bottom two rows are shadow replicas. Each of the colored bars indicates a unique resultant that must be transmitted in response to processing this most recent engagement request. Time is moving from left to right across this display, therefore the resultants shown from left to right indicate a sequence of activity occurring at each replica. There are four resultants for Auto_Special type engagement requests. Notice that the colored bars are taller for the primary replica. This indicates that it has actually transmitted the resultants. The half-height bars for the shadow replicas indicate they have verified that the primary replica has successfully transmitted the resultants. Notice that each replica either transmits or verifies the same set of resultants. This is an important insight given by this display. This indicates that the replicas are coming to the same conclusions about the processing steps that must be completed for this engagement request. They are "in the same state" with respect to this engagement request. If this display ever indicates colored bars that lack this "vertical harmony" among the replicas it would mean that they have inconsistent state.

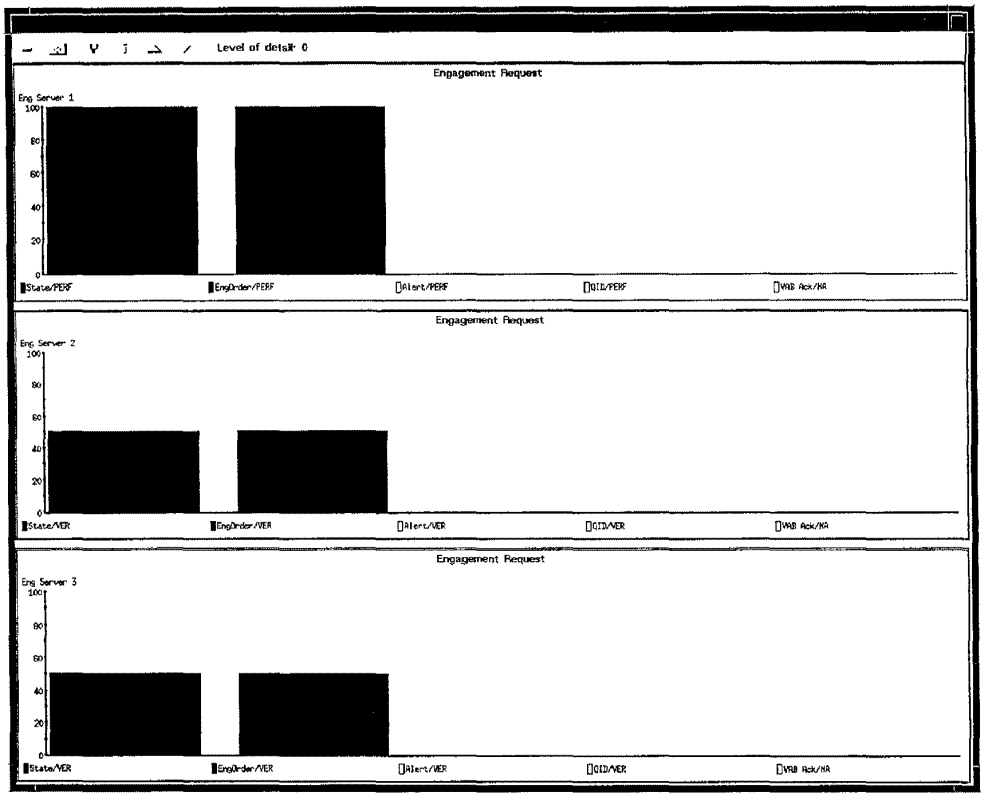


Figure 3.7.3-1 JEWEL Display

3.7.3.1 Fault Injection Control

a. Prior to Demo 98 the HiPer-D project did not demonstrate precise fault injection capabilities. HiPer-D has dealt primarily with process failures and, on a more limited basis, with hardware failures. Process failure was accomplished through the use of Ctrl-C and Unix level Kill signals. This did not provide the capability to inject a fault, resulting in a process failure, at a precise point in time or a precise location in a process. It was necessary to hand tailor a fault injection capability to provide this level of control. This was accomplished through the creation of a data file that specified the fault injection parameters. It also required application changes to read and respond to the fault injection parameters in this data file.

b. The data file contains the following information:

(1) Propagate True
(2) Replica 0 Trigger 2 Skip 2

c. The first line indicates whether automatic generation of a fault is enabled or not. A value of True for the Propagate variable specifies that fault injection is enabled. This allows the injection to be enabled and disabled at runtime. The second line provides the details of the fault injection. The Replica variable specifies which of the replicas is to fail; 0 is the primary replica and 1+ are Shadow replicas. This allows failure of any replica to be tested and demonstrated.

The Trigger variable specifies which location in the process is to generate the fault. There are several locations encoded in the Engagement Server component that correspond to unique Trigger numbers. This allows the server to proceed to the specified point in the processing before the fault is generated. The Skip variable indicates the number of engagement requests to skip over before the Replica and Trigger values become active. This allows finer control during testing and demonstration.

d. All Engagement Server replicas read this specification file at a ½ Hertz periodic rate. Currently there are five locations in the replicas where code has been modified to test for fault generation. These correspond to Trigger values 1-5. At each of these five test locations the following conditions must be satisfied before a fault is generated:

(1) This is the correct Replica number
(2) This is the correct Trigger location
(3) Fault injection Propagation is True
(4) The proper number of engagement requests have been Skipped since Propagation was enabled.

e. When all conditions are satisfied then a fault is generated. The fault is in the form of an Ada exception that propagates to the main procedure of the engagement server process. At this point a process failure is generated. While this technique is somewhat rudimentary in nature, it does allow a precise and repeatable fault injection capability. It can be used to fail a process at a user-specified location thereby allowing visualization and measurement of the resulting impact.

3.7.3.2 Fault Recovery and Performance Impact

a. During Demo 98 a fault was intentionally injected into the primary Engagement Server replica using the control described in the preceding section. This resulted in a primary replica failure that occurred during an engagement request originating from a SPY-declared Auto_Special contact. The SPY-declared Auto_Special engagement path is the most stringent timing requirement in the C&D Element. The time allowed between the initial SPY qualification and the engagement order generated to WCS is very short.

b. Figure 3.7.3.2-1 illustrates how this path, or timeline, corresponds to components in the HiPer-D system. There are four components involved in the SPY-declared Auto_Special path; Track_Control, Auto_Special, Engagement Server, and WCSSim. The green and red arrows indicate messages that flow between these components in executing this path. The wall time taken to process this engagement is shown as the bar with "**Begin Review Path**" and "**End Review Path**" at each end. The last activity in this path is the transmission of the engagement order to WCSSim, i.e. the red arrow in the diagram. The wall time to execute this path must not exceed the C&D timing requirement.

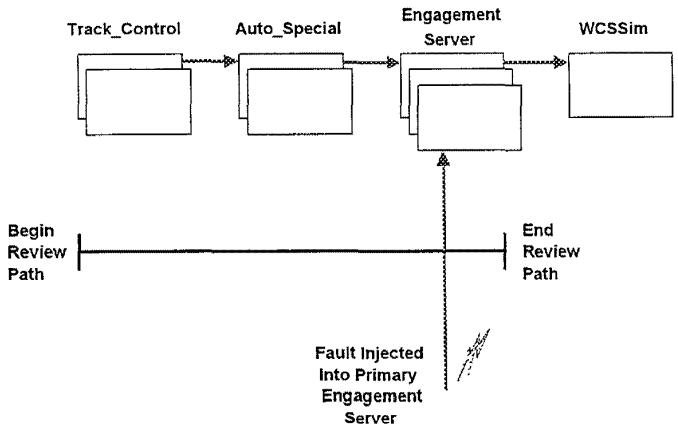


Figure 3.7.3.2-1 SPY-declared Auto-Special Review Path

c. The blue arrow indicates that a fault will be injected into the primary Engagement Server during this stringent path. Again, this will result in a process failure of that primary replica. This is by intent and allows analysis of two important attributes:

(1) The successful completion of the engagement request even during failure of the primary replica.

(2) The impact on performance in meeting the Auto_Special timing requirement.

d. The JEWEL display, previously described in Figure 3.7.3-1, allows insight into the first of these attributes. Figure 3.7.3.2-2 shows another image of this JEWEL display. It shows this display captured during a live Demo 98 execution shortly after the primary Engagement Server has been intentionally failed. The primary replica failed just before transmitting the engagement order to WCSSim; (the blue box would have been this resultant). Both shadow replicas detected this failure. The oldest shadow replica assumed the primary role. This is seen in that this shadow replica verified the first resultant from the failed primary replica, i.e. the red box, but it transmitted the remaining resultants after taking on the primary role (remember that verifications are half-height and transmissions are full-height). The other shadow replica remained in a shadow role even after recovery from the failure. It verified the transmission of all resultants regardless of whether they originated from the failed primary replica or the new primary replica.

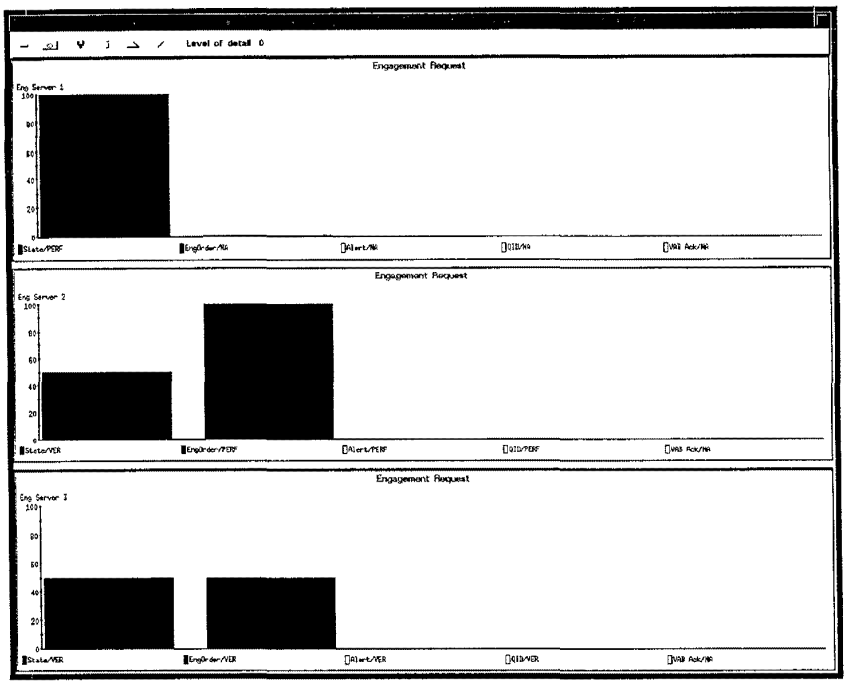


Figure 3.7.3.2-2 JEWEL Display Illustrating Failure of Primary Replica

e. This display showed that the current engagement completed successfully in spite of the process failure. This is an important feature of the Engagement Server in that it provides automatic continuation of an engagement in progress even if a replica is lost. This display also showed that only one resultant of each type was transmitted, i.e. no duplicates or lost messages. This is also important in that lost or redundant messages can create error conditions in other components that are expecting to see one and only one such message.

f. The second attribute is the ability to assess the performance impact on the Auto_Special path. Figure 3.7.3.2-3 shows an instance of the Auto_Special JEWEL display. It was captured during a live Demo 98 execution shortly after the primary Engagement Server suffered a process failure. There are five charts on this display. The upper four visualize aspects of the Auto_Special doctrine clients' periodic activities. The bottom chart that extends over the full width of the display is the only one of relevance to this particular discussion. Figure 3.7.3.2-1 showed the SPY-declared Auto_Special review path. The bottom chart on this JEWEL display in Figure 3.7.3.2-3 is a visual representation of the wall time required by the HiPer-D components to successfully process this path.

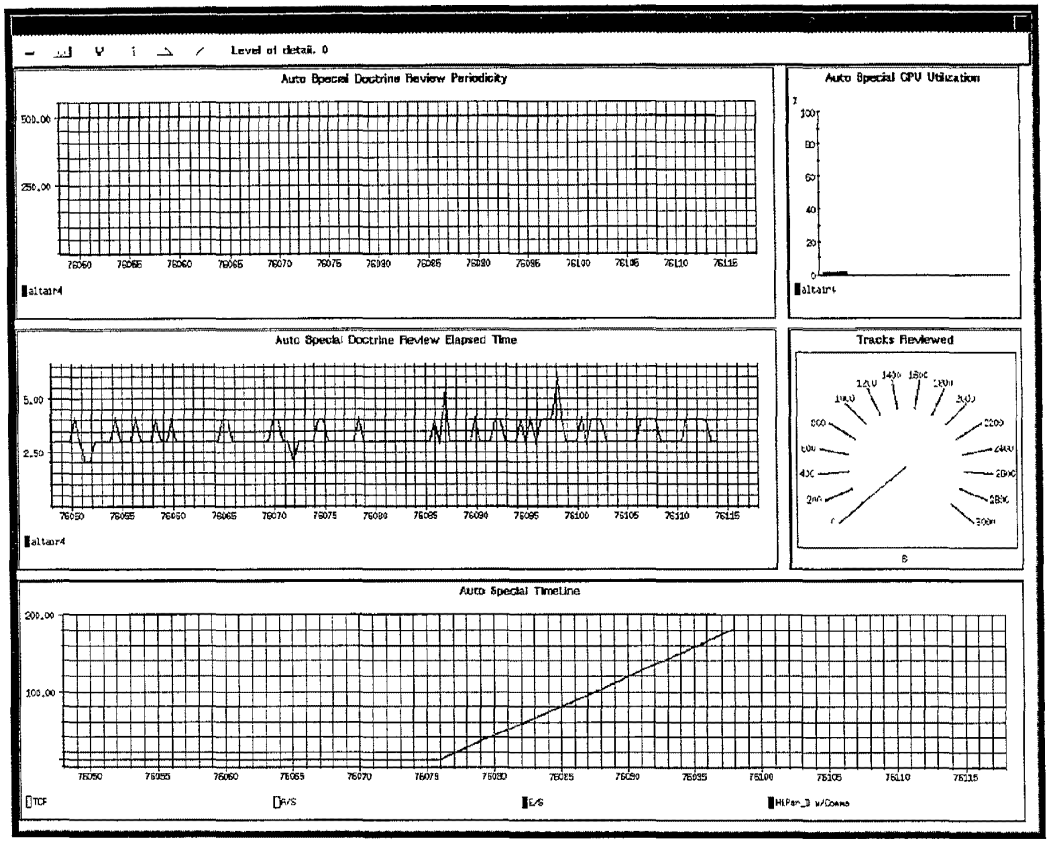


Figure 3.7.3.2-3 Auto_Special JEWEL Display

g. The C&D timing requirement is represented on this chart as 100%. In other words, as long as the HiPer-D components collectively take less than 100% to process a SPY-declared Auto_Special contact then the requirement is being met. Typically the HiPer-D components consume ~10-20% of the requirement, well below what is allowed. Notice that the blue line drawn from left-to-right across this chart rises sharply about halfway across the chart. This is the impact due to the primary Engagement Server failure. Prior to that failure the components were using less than 20% of the allowed time. This can be seen by the fact that the left-most scale shows numbers at 100% and 200%. The 100% mark is on the fifth gridline. The blue line initially starts below the first gridline, a value less than 20%.

h. At the point where the failure occurs the line increases to ~180%. This is a violation of the requirement as specified in C&D. But, three important points must be stressed here. First, there is not actually a C&D requirement specified for the SPY-declared Auto_Special path under failure conditions. HiPer-D has taken this stringent requirement and attempted to meet it even in the presence of process failures, an aggressive and challenging undertaking. Second, the visual impact and analysis of this performance hit is only enabled by the precise and repeatable fault injection capability tailored by HiPer-D. This stride is essential for analysis of both performance and resiliency of distributed systems. Third, if C&D is lost in AEGIS, Baseline 5, it must be restarted in the N+1, ACTS computer (a cold restart). It must progress through the complete

initialization phase before it is ready to process Auto_Special engagements; not counting the fact that the Auto_Special engagement in progress during the failure could be lost. The recovery time for this restart operation is significantly longer than the 100% mark specified for the requirement.

i. There is a hot restart mode in AEGIS that takes advantage of the shadow memory feature of the UYK-43 architecture. This failure and recovery mode in C&D is much faster than the cold restart mentioned above. Even so, it still could take up to 500% of the Auto_Special time we have been discussing to perform a recovery in this manner. With these points in mind it follows that 180% of the requirement during a loss of the primary replica is very good performance by these HiPer-D components. Nevertheless, it is one of the goals of HiPer-D to continue to analyze and improve recovery performance in the presence of failures.

3.7.3.3 Summary and Future

a. Adding fault tolerance to the Engagement Server component for Demo 98 created a completely fault-tolerant engagement path through the HiPer-D system. The Engagement Server design allows these replicas to keep the critical engagement state data consistent among all replicas. The engagement state data is potentially affected by messages from several communication groups making the design to guarantee consistency a challenging problem to solve. The new JEWEL display provides a window into this state and action consistency among the replicas. Finally, the precise fault injection addition allows repeatable fault testing as well as measurements of system performance.

b. In the coming year Engagement Servers will have the capability of coming on-line even in the middle of an engagement request. Currently, redundant servers can be added back into the system, but only when there is no engagement related activity. It is import to ensure that a new server receives the proper state as it comes on-line in order to know where in the processing sequence it should begin. This is an interesting problem if an engagement request is in progress. An additional factor is that multiple communication groups can be involved during the processing. Transference of the proper state to this new server under these conditions is a challenging endeavor.

3.7.4 Digital Call for Fire (CFF)

a. One of the enhancements to the HiPer-D testbed for Demo 98 was the development and integration of a remote interface to support simulation of an external digital CFF capability. The Demo 98 scenario was designed to demonstrate the expanded CFF capability for both scheduled and unscheduled digital calls for Naval Surface Fire Support.

b. This capability encompassed several major components and processing elements within the total system, including:

(1) The creation and utilization of a remote Forward Observer / Forward Air Controller (FO/FAC) simulation subsystem.

(2) Visual deconfliction of the air picture.

(3) The insertion of an OTH CFF track into the HiPer-D system, representing the target aim point of the CFF engagement.

(4) The transmission of engagement requests by the Tacfire Processor to the Engagement Server.

(5) The prosecution of the engagement by the NSFS Simulator (NSFSsim) with subsequent spotter adjustment(s) provided by the remote FO/FAC.

(6) Successful completion and termination of the CFF engagement.

c. Each of these functional areas required substantive new design, development or upgrading of the software base in the testbed. A high level graphical summary of the system is provided in Figure 3.7.4-1.

3.7.4.1 FO/FAC Subsystem

a. Figure 3.7.4.1-1 provides details on the FO/FAC components. The drawing highlights two areas:

(1) It illustrates the actual connectivity between the main testbed in Building 1500 and the remote CFF FO/FAC operator in Building 180, both within the NSWCDD complex.

(2) It also provides an indication of the CFF interface message flow between the two sites and supporting Tacfire/RDDL components that emulate the prototyped FO/FAC capability.

b. One of the objectives of the demonstration was the creation of a remote FO/FAC capability to simulate a spotter on land transmitting CFF requests and corrections digitally to a combatant supporting the land attack mission. To achieve a certain degree of reality, the following design decisions were made:

(1) The fixed format Tacfire message specification was selected in accordance with "Interface Specification for Maneuver Battalion Fire Support Element (Advanced Field Artillery Tactical Data System)" (FSSIS-IS-0093 Rev. A, dated 1 December 1993) as representative of this interface.

(2) The militarized AN/GRA-39 transceiver set was integrated into the remote FO/FAC environment.

(3) Supporting components, Scenario Injection Stimulator (SISTIM) and a Remote Digital Data Link (RDDL) provided both the CFF scenario mechanisms and the TCP/IP network protocol encoding of the CFF messages.

c. The result of these efforts provided the infrastructure approximation that represented a remote CFF operator in the HiPer-D Demo 98 testbed.

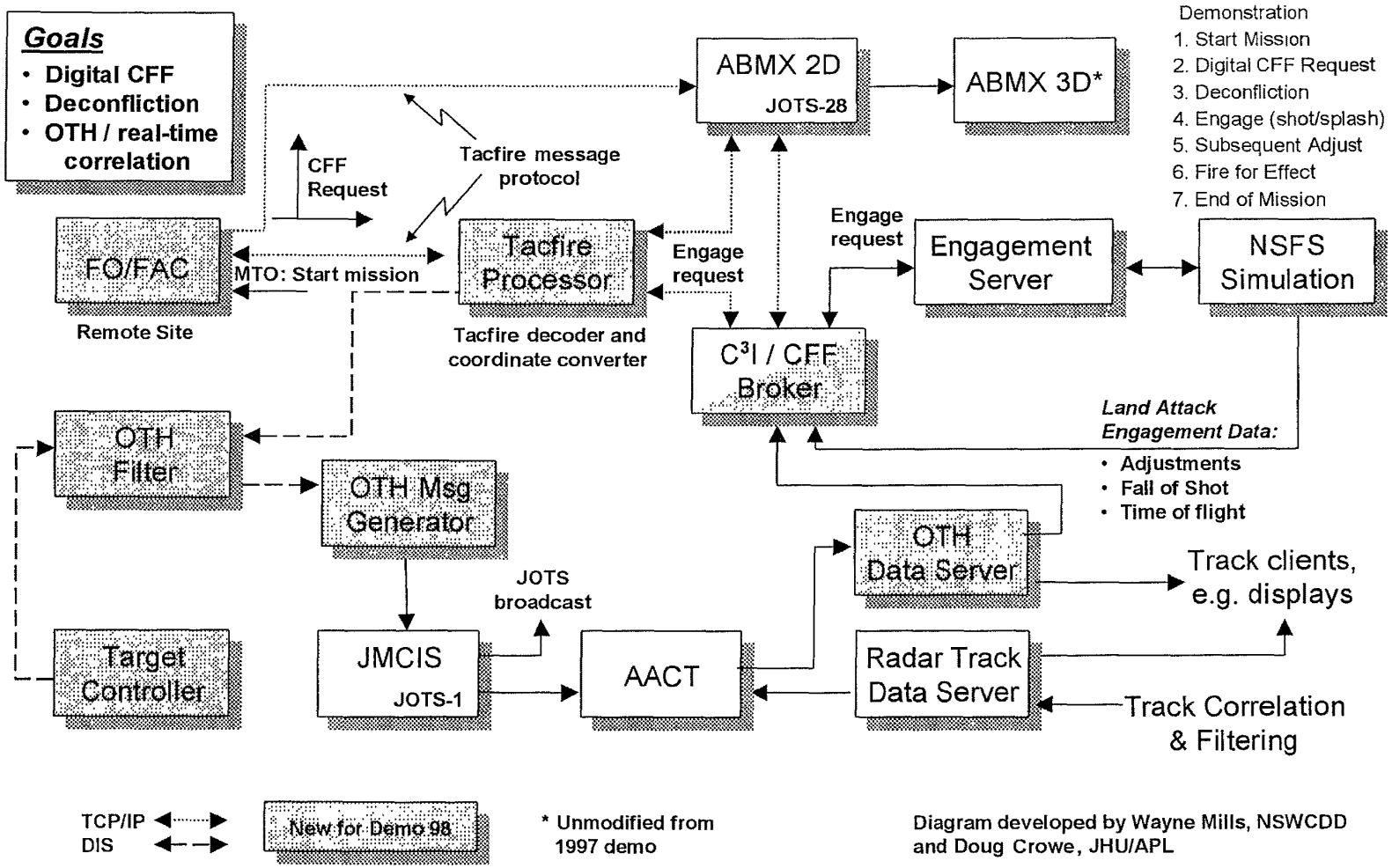


Figure 3.7.4-1 Digital Call for Fire Diagram

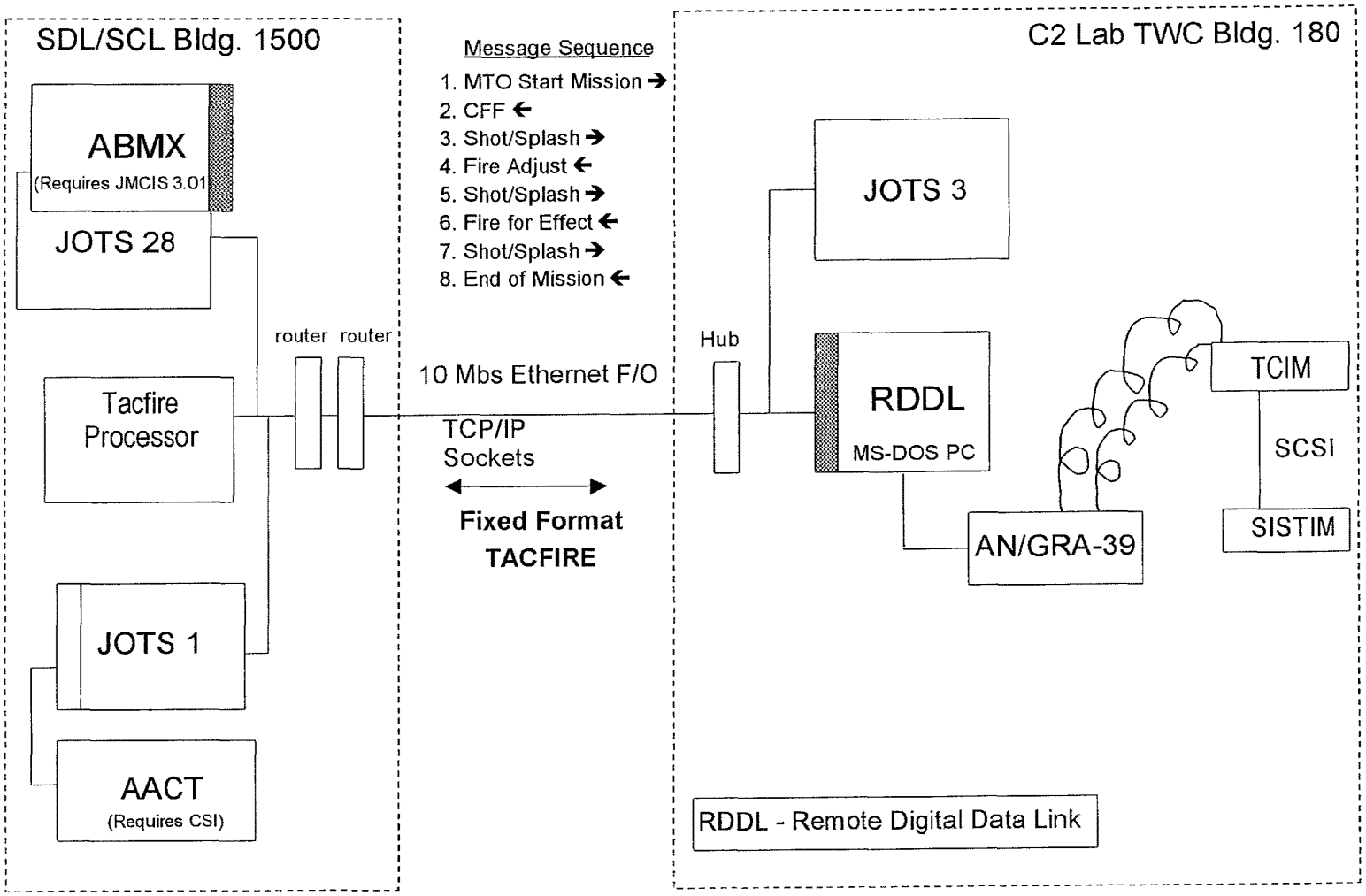


Figure 3.7.4.1-1 FO/FAC Subsystem Components

3.7.4.2 CFF Initiation Sequence

a. To facilitate the testing and demonstration of the CFF capability, the testbed-specific Message-to-Observer (MTO), Start Mission, was defined consistent with the Tacfire specification. This initiated each engagement and alerted the remote FO/FAC that the testbed was ready to conduct a test mission.

b. Upon receipt of the Start Mission message, the FO/FAC then selects and executes a defined CFF target scenario from the SISTIM. This results in the RDDL transmitting the initiating Tacfire CFF Request message. This flow initiates with the remote FO/FAC and is sent to the Advanced Computing testbed's Tacfire Processor and ABMX subsystem over a TCP/IP socket connection. At this point, a single land attack mission is in progress.

3.7.4.3 Visual Deconfliction

The Air Battle Management and Execution (ABMX) element supplied by the Naval Research Lab (NRL) was integrated into the testbed, primarily to provide a deconfliction facility. Upon receipt of the initiating Tacfire CFF Request message from RDDL, ABMX computes and plots the start and endpoint of the anticipated ballistic flight path of the land attack mission. This data is displayed in both 2-D and 3-D on the ABMX displays. In particular, the 3-D visualization, along with the receipt and presentation of the complete track picture from the GCCS-M (a.k.a. JMCIS) Jots-1 platform, allows rudimentary visual deconfliction of the air picture with the intended projectile flight path. In this version of the prototype, voice communications are used as the mechanism for communicating information should safety conflicts be detected.

3.7.4.4 OTH Track Injection

a. Concurrent with ABMX performing its deconfliction processing, the CFF Demo 98 design called for the injection of an OTH track into the system to permit the AEC to prosecute this target. Upon receipt of the CFF Request from RDDL, the Tacfire Processor decodes the UTM coordinate data of the Tacfire message and sends a DIS Entity State (ES) PDU packet to the OTH Filter component in order to initiate this sequence. The fundamental flow of data proceeds through the bottom left section of Figure 3.7.4-1:

OTH Filter → OTH Message Generator → JMCIS → AACT → OTH Data Server.

Ultimately, the OTH Data Server broadcasts the newly created CFF target track to all registered clients of this process.

b. This represents a departure from the T3 Demo (August 97) CFF approach in that in the previous milestone, a simulated fire control sensor was used to designate and create a target for this type of engagement. The Demo 98 redesign in this area is considered to be a more realistic representation of this capability.

3.7.4.5 CFF Engagement Transmission

a. Once the OTH track DIS ES PDU packet is sent, the Tacfire Processor transmits the Engagement Request message to the C^3I_Broker. This step begins the functional processing within the AEC system.

b. The C^3I_Broker receives the Engagement Request message and waits for the arrival of the CFF target track before it forwards the engagement. Using basic positional computations, the C^3I_Broker performs rudimentary comparisons of the engagement endpoint against the list of available OTH target tracks. Once a match is determined, the sequence proceeds with the transmission of the request to the Engagement Server. In the case that C^3I_Broker does not find a matching track for the requested target within a user-specified time period (defined in an adaptation data file), the engagement is terminated with a Cannot Comply (CANTCO) acknowledgement in the Time-of-Flight response message sent back to the Tacfire Processor.

3.7.4.6 Engagement Sequence

a. After C^3I_Broker sends on the request, the Engagement Server queues a CFF alert to the ASUWC (Anti-Surface Warfare Coordinator) submode position. The operator reviews the alert with the OTH track hooked, and approves the engagement of the target. This event causes an internal AEC message sequence of:

Manual Engage → Engagement Server → NSFSsim

which delivers the engagement request to NSFSsim. This simulator performs engageability and time-of-flight (ToF) calculations and provides the data directly to the C^3I_Broker. As this message is delivered, NSFSsim initiates a simulated land attack sequence representing the firing of the shipboard gun system. Figure 3.7.4.6-1 shows the ABMX 3-D graphical depiction of the projectile.

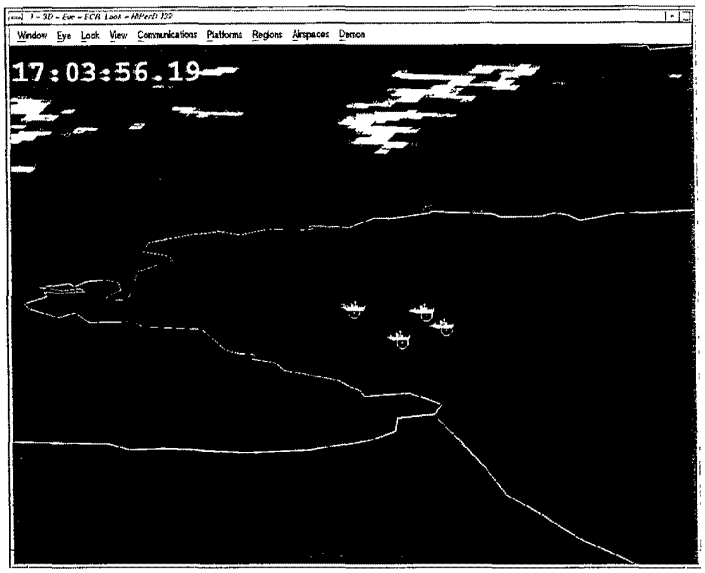


Figure 3.7.4.6-1 ABMX 3-D Display Showing Projectile Ballistic Flight Profile

b. C^3I_Broker responds to the Tacfire Processor indicating that an engagement is in progress, and sends along the ToF value. It follows this action with a similar notification to ABMX that allows this subsystem to set its displays to change the color and status of the engagement to "red," indicating that a firing is currently in progress (i.e., zone is "hot") as shown below in Figure 3.7.4.6-2.

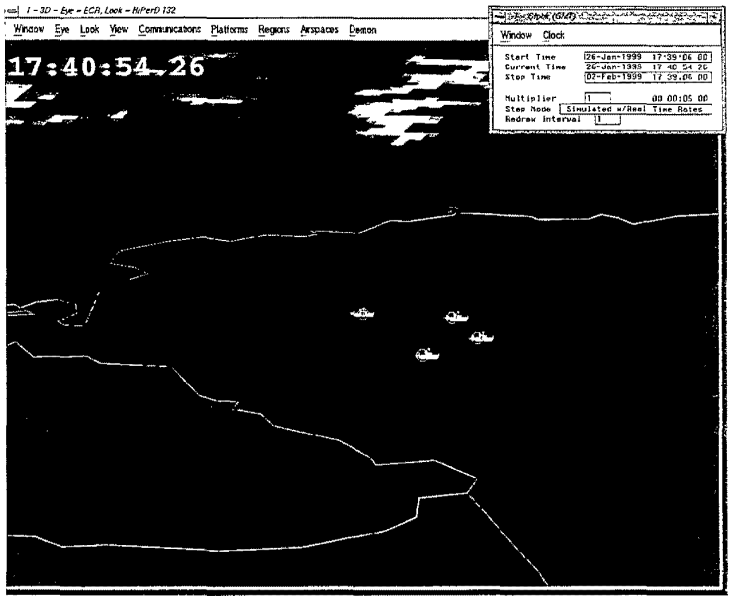


Figure 3.7.4.6-2 ABMX 3-D Display Showing "Hot Zone"

c. Tacfire Processor provides a 2-message sequence back to the FO/FAC:

(1) Indicating the "Shot" has just left the ship; and

(2) A "Splash" message 5 seconds prior to computed projectile impact.

d. These notifications permit the forward observer to assess fall of shot on target. The FO/FAC then provides adjustment, based on assessed error of the previous shot. This data is transmitted in a follow-on Spotter Adjust message to the Tacfire Processor which in turn forwards the data to the C^3I_Broker. C^3I_Broker sends the correction to NSFSsim who recomputes the required parameters, issues his reply back to the C^3I_Broker and commences the firing sequence. C^3I_Broker transmits the ToF data back to the Tacfire Processor who proceeds to issue the "Shot" / "Splash" 2-message sequence to FO/FAC.

e. This series of shot and spotter correction can be repeated as necessary in order to allow the FO/FAC to place the fall of shot on target. The corrections and number of spotter rounds are completely driven by the commands issued by the remote spotter simulated in building 180.

f. Once the FO/FAC is satisfied that spotter rounds are on target, he issues a fire-for-effect request. In Demo 98, this was done to allow multiple rounds to be fired at the CFF target in rapid succession, under remote spotter control, in order to destroy the mission objective. The specific messages are identical to the ones discussed above, with the only exception being that the final series of "Shot" / "Splash" messages back to the FO/FAC represent the starting and end-point times of the multi-round sequence.

g. Once the spotter is satisfied that the target is destroyed, he issues a Tacfire End-of-Mission (EoM) notification. This message permits both the ABMX subsystem and the AEC system to clear the engagement from their databases. The Tacfire Processor forwards the data to C^3I_Broker that sends an Engagement Termination message to Engagement Server. Engagement Server issues internal status messages to its clients and queues a Kill information alert to the ASUWC operator to complete the termination processing. At this point, the system is ready to receive another remote CFF request against a new target.

h. The defined sequence of events and messages is shown in Table 3.7.4.6-1.

Table 3.7.4.6-1 CFF Primary Message Processing Flow

	SOURCE	*DESTINATION*	*MESSAGE*
0	Tacfire Processor	FO/FAC	MTO: start mission
1	FO/FAC	Tacfire Processor, ABMX (info)	Tacfire CFF request
2	Tacfire Processor	OTH Filter	DIS entity state (Create Target)
3	Tacfire Processor	C^3I Broker	Engagement request
4	C^3I Broker (waits on receipt of OTH track report)	Engagement Server	NSFS land attack engagement request
5	Engagement Server (operator initiated via Manual Engage)	NSFSsim	Target engage request
6	NSFSsim	C^3I Broker	Land attack engagement data
7	C^3I Broker	Tacfire Processor, ABMX (info)	Time-of-flight (ToF) (MT-99)
8	Tacfire Processor	FO/FAC	Shot/Splash (2 msg sequence)
9	FO/FAC	Tacfire Processor, ABMX (info)	Spotter adjust (5-10 secs after hit)
10	Tacfire Processor	C^3I Broker	Subsequent adjust (MT-100)
11	C^3I Broker	NSFSsim	Shot adjust
...	***(repeat 9-11 then 6-8 as required for multiple spot adjustments)***		
12	FO/FAC	Tacfire Processor, ABMX (info)	Fire for effect
13	Tacfire Processor	C^3I Broker	Subsequent adjust (MT-100) (# of effect rounds in originating CFF message)
14	C^3I Broker	NSFSsim	Shot adjust
15	NSFSsim	C^3I Broker	Land attack engagement data
16	C^3I Broker	Tacfire Processor	ToF (MT-99) (for first effect rounds)
17	Tacfire Processor	FO/FAC	Shot
18	FO/FAC	Tacfire Processor, ABMX	End-of-mission (EoM)
19	Tacfire Processor	C^3I Broker	Engagement complete
20	C^3I Broker	Engagement Server	Engagement termination

3.7.5 Demo 98 Resource Management Scenario

a. The Resource Management efforts for FY98 resulted in the development of many capabilities and features that could have been demonstrated in Demo 98. Due to time limitations, however, it was decided to focus on only a handful of key capabilities. This section describes the Demo 98 Scenarios which highlighted several key Resource Management capabilities.

b. During the demonstration, the following three key capabilities were demonstrated:

(1) Fault Tolerance of Resource Management Components (survivability of Resource Management)
(2) Scalability of the AAW Doctrine Processes (load-balancing for handling increasing tactical load and changing mission requirements)
(3) Fault Tolerance of the AAW Doctrine Processes (ability to continue to meet mission requirements in the event of software failures)

c. In addition to the demonstrated capabilities, the following capabilities were also shown but not specifically focused on:

(1) Monitoring and control across all UNIX platforms in the testbed.
(2) Startup and shutdown of infrastructure components (RM, displays, etc...).
(3) Initial host selection by RM for selected applications.
(4) Startup, shutdown, and configuration of applications based on QoS Specifications.

d. The remainder of this section discusses each of the Resource Management capabilities and scenarios that were demonstrated during Demo 98.

3.7.5.1 Overview.

a. The Resource Management portion of the demonstration was the concluding section of Demo 98. The capabilities demonstrated were broken down into four main sections.

b. For the first section of the demo, some of the fault tolerant capabilities of the Resource Management components themselves were demonstrated. Since Resource Management is envisioned as controlling the configuration and allocation of other shipboard systems, in order to be effective, it must be survivable. This year, restart fault-tolerance for most of the Resource Management components was implemented; almost any component of the architecture and infrastructure can be faulted and restarted with the exception of the Resource Manager itself. In particular, multiple host monitors, being faulted and restarted, were demonstrated. The key point was to demonstrate the kinds of survivability capabilities needed by the Resource Management components to handle and recover from hardware or software failures.

c. For the second part of the demonstration, the AAW AutoSpecial and SemiAuto Doctrine processes were scaled up based on detection of overload conditions by Resource

Management. The doctrine processes themselves are designed to perform load balancing between replicas. This capability was demonstrated as well as demonstrating Resource Management detecting the existence of overload conditions (based on increasing tactical load), and deciding when and where the scaled up doctrine processes should be started. The key point was to demonstrate the ability of the Resource Management components to dynamically detect overload conditions and effectively scale up in order to continue to meet mission requirements.

d. During the third section of the demo, several AAW Doctrine processes (AutoSpecial and SemiAuto) were faulted in order to demonstrate fault detection and automatic restart of the AAW Doctrine processes by the Resource Management component. During this part of the demonstration the reconfiguration and load-balancing capabilities of the AAW applications were highlighted. (Also, earlier in the Demo, during the ATWCS section, the ability of the Resource Manager to detect the failure of an ATWCS application, the LC-RT component, and automatically restart it was demonstrated.) The key point was to demonstrate the ability for the Resource Manager to reconfigure the system to continue to meet mission requirements even in the event of software failures.

e. The fourth section of the demo involved ramping up the track load to over 7000 tracks and scaling up to five copies each of the AutoSpecial and SemiAuto Doctrine processes. While this was being done, the Resource Management sections were summarized, and the overall demo summary was presented. The demo concluded, as it has in previous years, by demonstrating that even at extremely high system loads, the AutoSpecial engagement timelines still fell well within spec.

f. The next three sections look in detail at each of the three main Resource Management capabilities that were demonstrated:

(1) Fault Tolerance of Resource Management Components
(2) Control of Application Scalability of the AAW Doctrine Processes
(3) Application Fault Detection and Recovery

3.7.5.2 Fault Tolerance of Resource Management Components

a. The first section of the demonstration involved faulting several (arbitrarily selected as five) UNIX host monitor components of the Resource Management infrastructure. The Host Monitors selected to be faulted were running on a mix of SGI, SUN, and HP workstations. Before the Host Monitors were faulted, they were hooked on the Host Display (Figure 3.7.5.2-1), and data being collected by the Host Monitors was displayed in real-time on the Graph Display (Figure 3.7.5.2-2).

b. When the Host Monitors were faulted, depending on the timing of the faults, several of the hooked boxes on the Host Display would turn gray for a short time and then turn back to blue when the Host Monitors had been restarted. (Typically, the Host Monitors would be restarted and reconnected into the Resource Management system within a couple of seconds.) In other cases, depending on the exact timing, the restart of the Host Monitors and reconnection of the Host Monitors back into the Resource Management Infrastructure would result in no color

change, which indicates that from the display's perspective, no interruption of data was even noted.

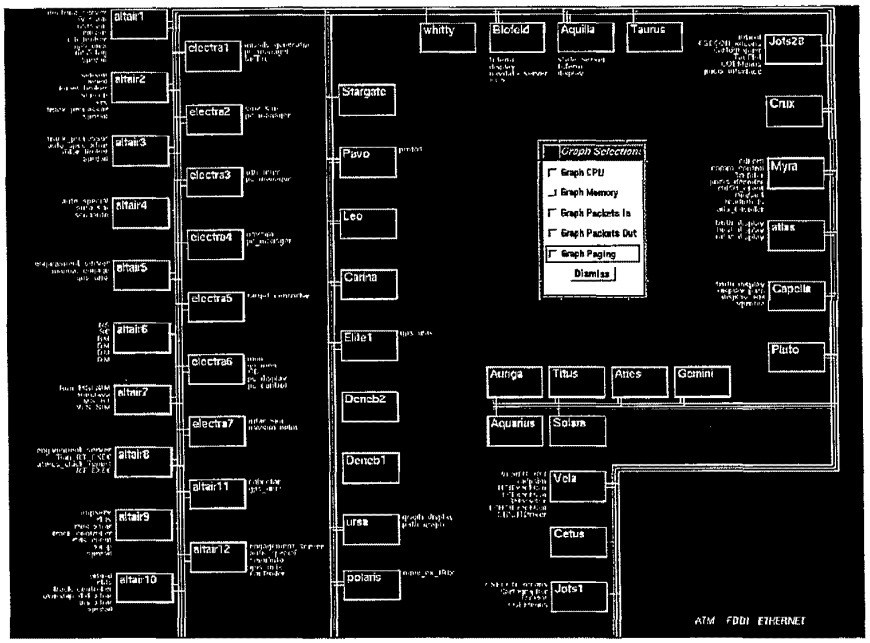


Figure 3.7.5.2-1 Hosts can be hooked for display.

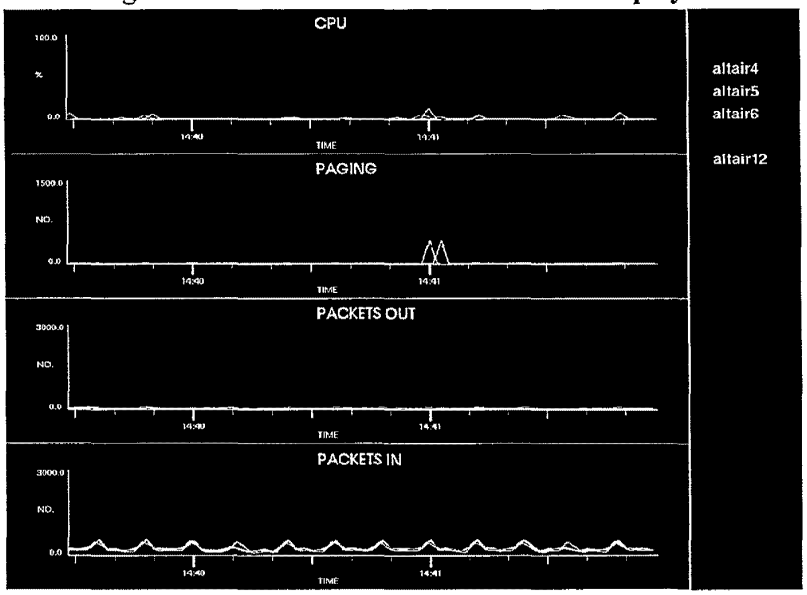


Figure 3.7.5.2-2 Performance data plotted from hooked hosts.

c. When the Host Monitors were faulted the application failure was detected by the Program Control agents. The agents then passed this information to the Program Control components which informed the Resource Manager that the Host Monitors had abnormally terminated. At this point the Program Control component also informed the Program Control Display (Figure 3.7.5.2-3) that the Host Monitors had been faulted and the corresponding element on the Program Control Display was turned red to indicate the failure.

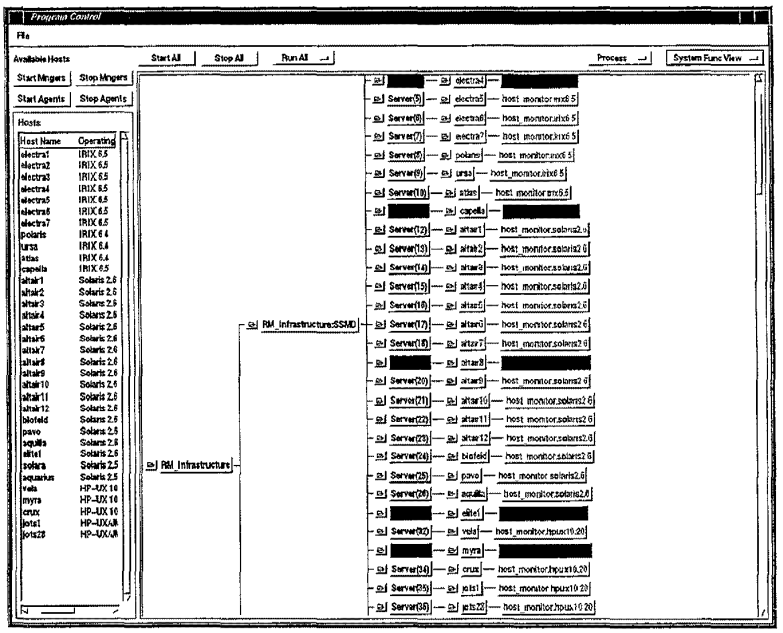


Figure 3.7.5.2-3 Program Control Display with faulted host monitors

d. The Resource Manager then determined from the QoS Specification information that the Host Monitors were restartable and should be restarted. The Resource Manager then ordered Program Control to restart the faulted Host Monitors. The Program Control component then consulted the QoS Specifications to determine exactly how to start the Host Monitor, and passed the order to the Program Control agents. The Program Control agents then started the Host Monitors. Once the Host Monitors successfully began executing, the Program Control agent informed Program Control that the Host Monitors had been restarted. Program Control then informed the Program Control Display that the Host Monitors were back up and the corresponding element on the Program Control Display was turned back to green. Program Control also informed the Resource Manager that the Host Monitors were successfully restarted. At the same time the Host Monitors were reconnecting back into the Resource Management Infrastructure and once again began sending data which was reflected on the Host Displays and the Graph Display.

e. All of this occurred extremely quickly and automatically with no operator intervention. The Program Control Display typically blinks red and back to green so fast that it is hard to be seen. On the other hand, the Graph Display typically shows a second or two of missed data, and the Host Display may show the Host Monitor being down for several seconds.

APPLICATION :	HostMonitor	EVENT # : 1
EVENT :	Application Fault PID :3924 ON HOST altair8	
EVENT TIME :	14:50:33.0278	
ACTION :	Application Restarted PID :3957 ON HOST altair8	
ACTION TIME :	14:50:33.0290	14:50:33.1975
RESPONSE TIME :	0.0012	0.1697

Figure 3.7.5.2-4 Resource Manager Decision Display response times for fault/restart

f. The exact fault detection and recovery timing can be seen on the Resource Management Decision Review Display (Figure 3.7.5.2-4). This display typically shows that from the time the fault is detected and reported to the Resource Manager to the time the Resource Manager decides what to do usually takes between 0.2 and 2.0 milliseconds. The display also typically shows that from the time the fault is detect to the time the Host Monitor is restarted and begins executing is typically 0.1 to 0.3 seconds. These are extremely fast fault recovery times, however they do not include the time required by the application to reinitialize and rejoin and reconnect back into the rest of the system. Hence, it sometimes appears that from the Host and Graph Displays that several of the Host Monitors are down for several even though they have actually been restarted almost immediately.

g. The key point of this section of the demonstration is to show the types of fault recovery capabilities needed in order for the Resource Management components to effectively handle and recover from hardware and software failures.

3.7.5.3 Control of Application Scalability.

a. For the next section of the RM demonstration, Resource Management monitored and detected AAW Doctrine process overload conditions, and responded by scaling up additional load-sharing replicas of the processes to being the timing requirements back within specifications.

b. The demonstration began with a background track load of about 500 tracks. As shown on the Jewel Multi-AutoSpecial Display (Figure 3.7.5.3-1), there was one copy of the AutoSpecial doctrine process running which was reviewing all of the tracks on a periodic basis to determine if any of the tracks met engagement criteria. The AutoSpecial doctrine review time for the single AutoSpecial typically takes about 50ms to complete. Also, on the Path Display (Figure 3.7.5.3-2), multiple copies of the Track Processor which forms the tracks in the system, multiple copies of the Radar Track Data Server (RTDS) which distributes the track data to clients, 1 copy of the AutoSpecial Doctrine Process which periodically (about every half second)

checks the tracks against doctrine criteria and if the criteria is met sends out an engagement request, multiple copies of the Engagement Servers, and 1 copy of the WCS (Weapon Control System) simulator which actually performs the engagement are seen.

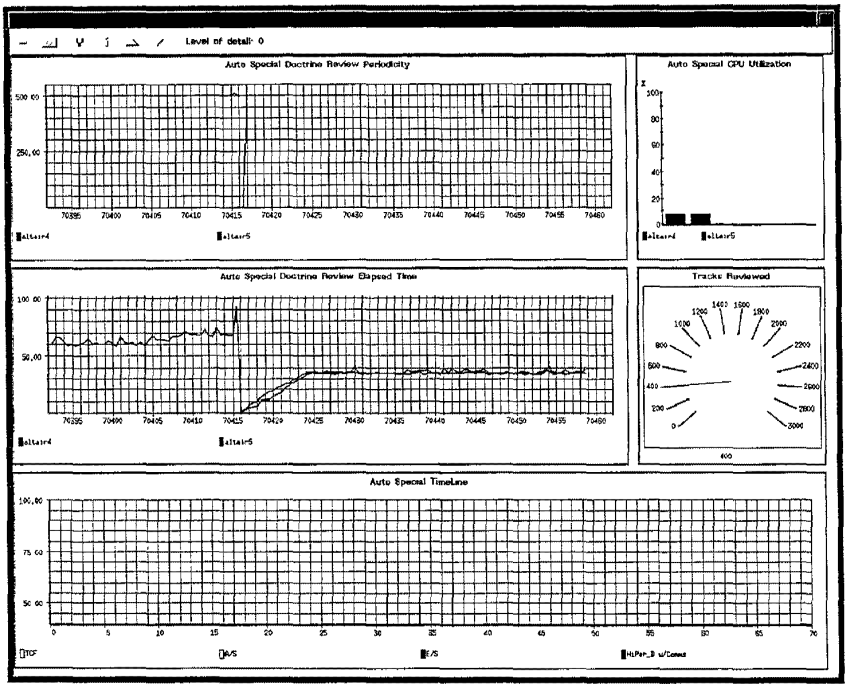


Figure 3.7.5.3-1 Jewel Multi-AutoSpecial Display

c. The doctrine review time requirement for the AutoSpecial doctrine process, as specified in the QoS Specifications, is 65 ms. As an additional 200 tracks are entered into the system, the doctrine review time steadily increases until it is violating the 65 ms threshold defined in the Specifications. The Resource Manager is monitoring the doctrine review times and when a certain number of samples within a sliding window (i.e., 10 of 20 samples) exceeds the threshold, the Resource Manager orders an additional copy of the AutoSpecial process to be started. As the second copy joins the system, the track load is redistributed among the two replicas, with each replica handling approximately half the track load, and the Multi-AutoSpecial Display (Figure 3.7.5.3-3) shows that the review time drops from a high of about 80 ms to about 40 ms for each of the replicas.

d. Also, as the new replica is started, the new AutoSpecial application shows up in yellow on the Host Display when it is first started (to indicate that a new process has been detected). The new copy of AutoSpecial also shows up on the Path Display.

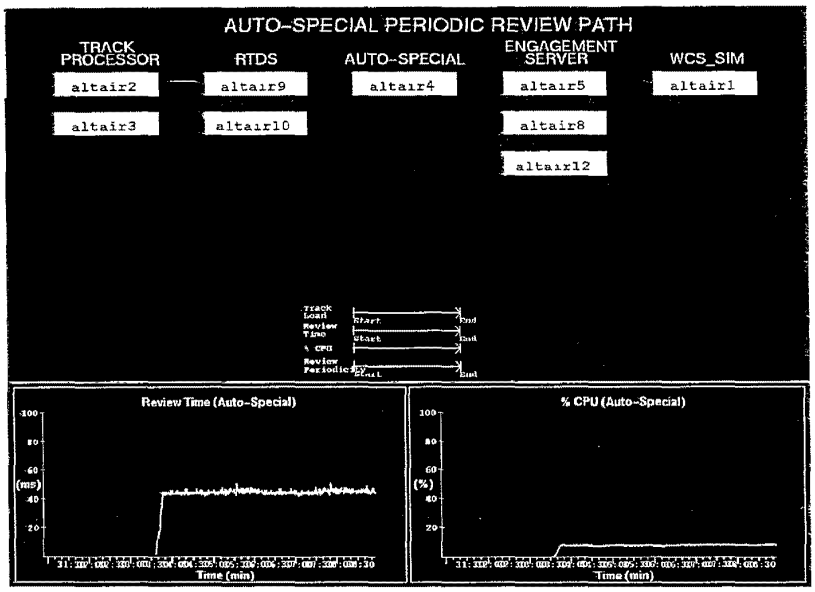


Figure 3.7.5.3-2 Path Display showing Auto-Special path

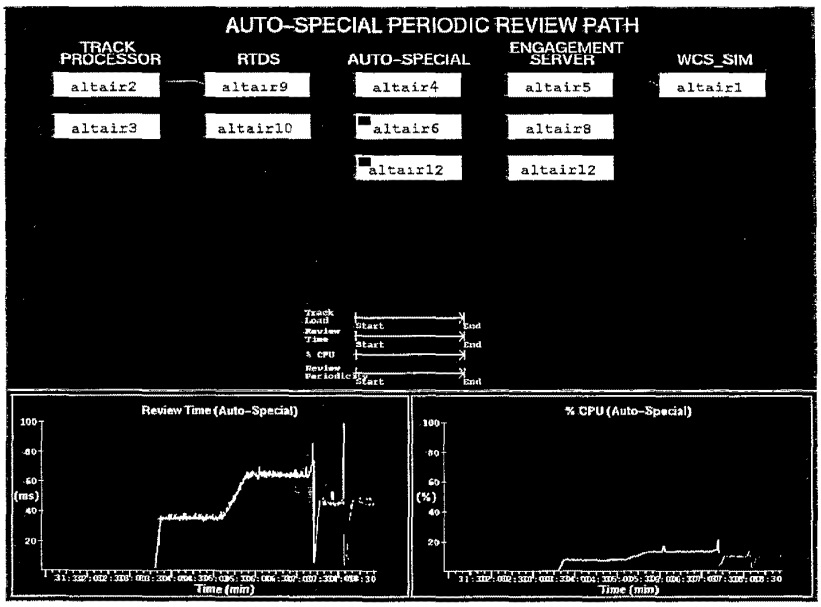


Figure 3.7.5.3-3 Path Display showing Scale-Up

e. To accomplish this the AutoSpecial application sends out instrumentation events (via Jewel) indicating how long it is taking to review its track load. These events are correlated and forwarded to the Path QoS Managers which determines (using a sliding window algorithm) whether or not the AutoSpecial doctrine review time requirement (as read from the QoS Specifications) is being violated. If a violation is detected and the AutoSpecial application is

specified as scalable (in the QoS Specifications), the Path QoS Managers inform the Resource Manager that the application is overloaded, and the Resource Manager in turn attempts to find the best host on which to run the new application. If a suitable host exists, the Resource Manager orders Program Control to start up a new copy of the application on the specified host. Program Control then starts up the new replica which joins into the system and triggers the RTDS's to redistribute the track load between the replicas.

f. The Resource Management Decision Review Display shows that the typical time from when an overload is detected to the time the Resource Manager decides where to scaleup an additional replica takes approximately 1.0 to 2.0 milliseconds. Typically, from the time the overload is detected to the time that the new replica is actually started and begins executing is about 0.1 to 0.3 seconds. Once again, as was noted for the faulted Host Monitors, application startup times are extremely low. However, from the Jewel data on the Multi-AutoSpecial Display, it is shown that it takes up to several seconds for the application to initialize and join into the track client group and several additional seconds for the redistribution of the track load to occur.

g. On the Resource Management Decision Review Display the gold bar represents the host where the replica was selected to be started (i.e, the host with the "best" host load score). The other bars indicate the next best hosts that could have been selected. As can be seen on the display, the aggregate host load score is a roll-up score based on CPU, network, and paging scores. The host load data being used is being generated by the Resource QoS Monitor components which is receiving the "raw" data value from History Servers (and indirectly from the Host Monitors).

h. At this point in the scenario, an additional 700 tracksare entered, which once again pushes the AutoSpecial review times above the 65 ms threshold and triggers a second AutoSpecial scaleup. Also, the SemiAuto Path Display, (Figure 3.7.5.3-4), shows that there is a single copy of the SemiAuto Doctrine process. As the SemiAuto review time approaches and then exceeds the 20 ms SemiAuto review time threshold (as defined in the QoS Specifications), the Path QoS Managers detect the overload and the Resource Manager triggers a scaleup of the SemiAuto Doctrine process. (The mechanism by which the SemiAuto scaleup occurs is identical to that of the AutoSpecial scaleup.)

i. What this demonstrates is the ability of the Resource Manager to dynamically detect application overload conditions and effectively scale up additional load-sharing replicas in order to continue to meet mission requirements.

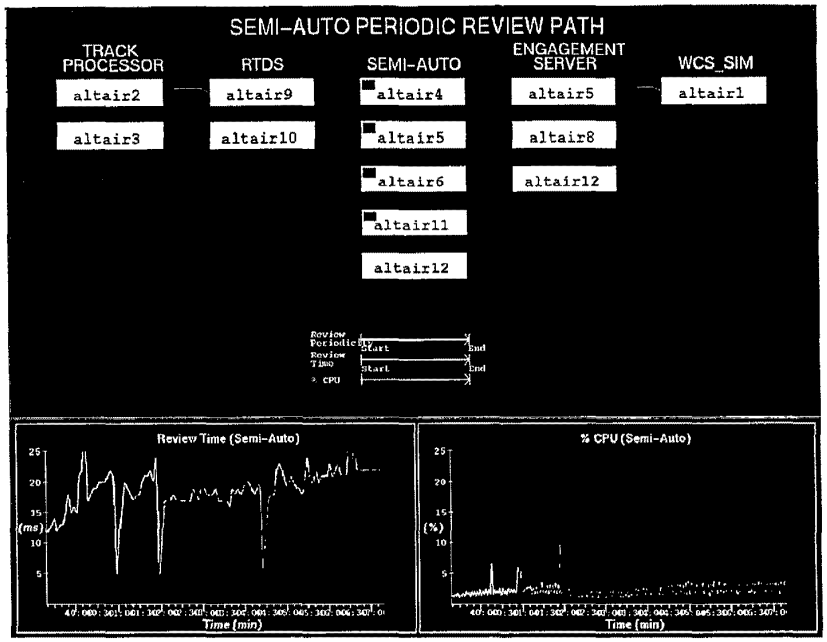


Figure 3.7.5.3-4 Path Display showing Semi-Auto Scale-Up

3.7.5.4 Application Fault Detection and Recovery.

a. In the third segment of the RM demonstration, AutoSpecial and SemiAuto Doctrine processes were faulted and automatically restarted by the Resource Manager. This section of the demonstration highlighted the ability of the Resource Management components to both detect application faults and to automatically determine where to restart the failed application.

b. For this part of the demonstration, there were three copies of the AutoSpecial Doctrine process running and two copies of the SemiAuto Doctrine process running. One of the copies of AutoSpecial was faulted and the Resource Manager detected the fault, determined that the application should be restarted, decided where to restart the application, and restarted the application. When the AutoSpecial application was restarted, the track load was redistributed properly and the review times remained at about the same times as before the fault.

c. During the fault and restart, the faulted application turned red on the Host Display (to indicate loss of the application), and the process turned gray on the AutoSpecial Path Display. On the Jewel Multi-AutoSpecial Display, the bar and line color corresponding to the faulted copy went away (and the colors assigned to the remaining copies may have been remapped). When the new AutoSpecial application was restarted, the new application appeared in yellow on the Host Display, and the new copy of AutoSpecial appeared on the Path Display. Also, the new AutoSpecial was assigned a new color on the Jewel display and the data was plotted.

d. All of this happened very quickly as determined by the timing information on the Resource Management Decision Display. The fault detection and recovery times are equivalent to the response values in the previous segment, typically about 1.0 to 2.0 milliseconds from the time the fault was detected until the Resource Manager decides where to restart the application,

and typically between 0.1 to 0.3 seconds from the time the fault was detected until the application was restarted and began executing. Again, the same caveats as in the previous sections apply; these times do not include the time needed for the application to initialize itself and join back in which the rest of the system.

e. Once again, the sequence of what happened is that the application fault was detected by the Program Control agent. The Program Control agent then informed Program Control. Program Control in turn informed the Program Control Display which turned the corresponding display element red. Program Control also informed the Resource Manager of the application failure. The Resource Manager then determined from the QoS Specification that the application was restartable and should be restarted. The Resource Manager than determined from the QoS Specifications the list of hosts where the application could be restarted. The Resource Manager then picked the "best" host based on host load scores provided by the Resource QoS Monitor, and sent the order to restart the application to Program Control. Program Control then sent the order to the Program Control agent which started the new application. Once the application had successfully begun executing, the Program Control agent informed Program Control that the application had been started. This information was then passed to both the Program Control Display (where it was used to update the corresponding display element) and to the Resource Manager (to verify that the order had been successfully enacted).

f. After the AutoSpecial application had been faulted and successfully restarted, a copy of the SemiAuto Doctrine process was then faulted. (The same Resource Management detection and recovery mechanism that was used for the AutoSpecial fault and restart was used for the SemiAuto fault and restart.) The SemiAuto doctrine process was successfully restarted with the same typical response times.

g. This part of the demonstration showed the ability for the Resource Manager to quickly reconfigure the system to continue to meet mission requirements in the event of software failures.

3.7.5.5 Summary.

a. Many features and capabilities of the enhanced Resource Management architecture were shown during the demonstration. In particular, significant improvement in fault detection and response times were demonstrated. In the FY97 T3 Demo, fault recovery times were in the range of 5 to 12 seconds; they are now typically in the sub-second range. First-step fault tolerant capabilities of the Resource Management infrastructure itself was demonstrated. Also, improved monitoring and control capabilities were shown; building this part of the infrastructure has been a key goal for this year since extensive near-real-time monitoring and control capabilities are essential for effective management of distributed mission-critical systems. The utility and effectiveness of our QoS Specification has also been demonstrated; the QoS Specifications are currently integrated throughout the Resource Management architecture.

b. Application fault detection and recovery capabilities have also been demonstrated. This was shown earlier in the demonstration with the restart of the failed ATWCS LC-RT component and was shown in the RM part of the demonstration using the mission-critical AAW

Doctrine processes. Host selection for the restarted components, based on host load metrics within constraints defined in the QoS Specifications, was demonstrated.

c. Resource Management detection of application overload conditions and Resource Management control of application scalability in order to continue to meet mission requirements was also demonstrated. This was demonstrated using the AAW Doctrine processes using review time thresholds defined in the QoS Specifications.

4.0 LESSONS LEARNED

4.1 CORBA Plan Server Lessons Learned

The goals to be achieved by investigating the CORBA technology were to:

(1) Gain first hand working knowledge of the CORBA technology standard.

(2) Gain experience in developing code that uses many of the key features that developers need to know to make effective use of CORBA.

(3) Gain experience in Object-Oriented distributed computing paradigm.

(4) Gain experience in "wrapping" legacy systems to use CORBA technology.

(5) Keep abreast of the latest CORBA standards and state of the market.

4.1.1 Advantages Offered by the CORBA Technology

CORBA is an object-oriented distributed system integration technology standard. Many distributed applications operate by passing messages. Message passing in distributed computing is very similar to method invocation on an object in object-oriented programming. CORBA offers distributed applications with many of the same benefits that object-oriented design provides to non-distributed computing; mainly reusable software, encapsulation, inheritance, as well as portability and extensibility. A primary objective of CORBA is to enable developers to program distributed applications using familiar techniques such as method calls on objects. In order to do this, CORBA significantly raises the level of abstraction, so that programmers do not have to deal with the low-level communication details. This results in ease of use for the programmer. However, keep in mind that this ease of use in programming is gained at the expense of some control and performance that is available at the lowest levels of communications programming.

4.1.2 CORBA Learning Curve

CORBA is more than just another middleware, it is an entire object-oriented distributed computing infrastructure with many components; for example; an ORB, IDL compilers, implementation repository, interface repository, Naming Service, Events Service, daemons, libraries, etc. just to mention a few. There is a substantial learning curve involved in effectively and proficiently use CORBA to design and build systems. This includes learning about object-oriented design and distributed object computing. In addition, depending on the CORBA implementation that is being used, there will be vendor specific features that must be learned.

4.1.3 Difficulties with Legacy Systems:

Many large, complex distributed computing systems already exist, many of which were not designed using object-oriented techniques. As a result, the task of migrating these systems from their existing middleware infrastructure to CORBA will be substantial. Some of the components may not be able to be treated as objects. It is important to keep in mind that it does not necessarily make sense to force an object-oriented middleware (such as CORBA) on non-object-oriented applications. Also, in migrating existing systems to CORBA, or even in designing and building systems from scratch, there are new and challenging design issues that will arise and must be addressed when an object-oriented middleware is used.

4.1.4 CORBA Specifications versus Available ORB Implementations

The OMG, which is the organization that oversees the adoption of new CORBA standards, states that on average, the length of time to approve and adopt a new specification is from 12 to 15 months. This is probably on the optimistic side. In reality, it can take much longer. In addition, availability of CORBA products that implement the specifications lag by many months and even years, if implemented at all. To be considered CORBA compliant, vendors must offer the CORBA core features. Although the elements that make up the CORBA core specification are very useful, they basically constitute a framework, and many of the additional features and services specified by the standard are not mandatory. Many of these optional services and features could be considered essential for developing large complex distributed systems. Initially, ORB vendors provide the features that most customers want and it varies as to which optional specifications are implemented. Also, because of the length of time required to adopt new standards, many vendors will offer proprietary solutions for features demanded by the market. As a result, different ORB vendors provide value-added services and functionality to the core in different ways which limits the portability of CORBA based applications. These proprietary features may even persist after standards are approved. Often these proprietary features provide important functionality and cannot be ignored when selecting an ORB.

4.1.5 CORBA in Perspective

a. For most large-scale, complex, distributed computing systems, it is highly unlikely that there will be a single middleware product that can provide all of the services needed with the required quality of service and performance. Many systems will most likely require a combination of different middleware products — both object-oriented and non-object-oriented — using each where it makes the most sense. Also, even if an object-oriented middleware such as CORBA is used, it may very well be that a single ORB will not be available that meets all of the requirements. Therefore, the system may have to use multiple ORB implementations that provide different features and levels of performance. The CORBA standard does provide for interoperability between different ORBs. Basically, in designing large, complex, distributed systems, CORBA should be considered along with message-oriented middleware and other object-oriented middleware.

b. For Demo 98, the Doctrine/Plan Server was modified to use CORBA to receive requests (via the ORB) from the clients for the weapon doctrine database. In addition, the ability for non-CORBA clients to request the weapon doctrine data using Isis was left intact. The clients were modified to use CORBA to make their requests (via the ORB) to the Doctrine/Plan Server by invoking the IDL "interface" operation getWeaponDoctrine on the CORBA Doctrine/Plan Server object. The ORB is responsible for ensuring that this invocation results in a call to the corresponding remote CORBA Doctrine/Plan Server object method, and returns the results to the client.

c. This experiment proved the feasibility of using CORBA in conjunction with group communication middleware such as Isis and provided insight into the practical aspects involved in using CORBA with legacy systems. Also, by choosing a non-critical area without strict real-time characteristics, it was possible to demonstrate successfully the use of the CORBA technology without affecting the overall system performance. The CORBA implementation (or ORB) used in this experiment was IONA Technologies Orbix MT 2.3c for SUN/Solaris2.6 which is CORBA 2.0 compliant. However, since CORBA is a standard, the lessons learned and knowledge gained should be applicable to other ORBs that follow the standard.

4.2 CORBA TNS Lessons Learned

a. The current CTNS architecture occasionally has difficulty shutting down all CTNS processes during HiPer-D shut down. Experimentation led to the discovery that the GMS communication layer thread (GMS_listen_thread) created for the CTNS_Broker application is often the receptor of the RUN.csh "termination" signal. This thread is created before an application's main function is executed, so that the application cannot control what action takes place when it receives a signal. The following conditions were observed:

(1) When CTNS_Broker (parent process) is started in an xterm, it and CORBA_TNS_Bridge (child process) both shut down most of the time. Occasionally, the child process does not shut down. This seems to occur whenever GMS is doing a great deal of logging.

(2) When the parent process is executed without an xterm, the child process does not shut down at all.

b. The xterm is not being utilized, except for allowing the parent and child to shutdown, because no output is being generated to the xterm by either the parent or child process.

c. A related issue has to do with the Orbixd debug modes. The default mode is non-silent, resulting in some output. The debug mode can also be specified in the application code. If a non-silent mode is specified in the code, launching Orbixd in silent mode does not override the mode specified in the code. If Orbixd is running with a non-silent debug mode, a destination for the output must be specified; otherwise CORBA applications will crash, with no indication why.

d. The Orbix version 2.2 documentation describes the ability to have different users launch and access a CORBA server, regardless of which user registered the server (using the putit utility) with the Orbix daemon. Supposedly, this was possible by using the chmodit utility to allow universal access to a server. As it turns out, this never seemed to work. Whichever user registered a server using putit had to be the user to launch the client application. This meant that each time HiPer-D was run, the user had to register the server, and then unregister the server when the run was complete. Otherwise, no other users would be able to run HiPer-D successfully. This problem was discovered just prior to the first scheduled on-site CORBA TNS test and delayed on-site testing for CTNS. Fortunately, the Orbix daemon under version 2.3 comes with an option to manage an unregistered server, so that use of the Orbix putit utility can be avoided.

e. Using a Naming Service is the CORBA-compliant procedure for connecting a client and a server. Orbix provides different versions of its Naming Service. The older, free version (OrbixNames version 1.03) was requested from Orbix, but there were substantial delays in receiving it. Because CORBA is not widely used on HiPer-D, the delays in receiving OrbixNames, and the difficulty in migrating from version 2.2 of Orbix to version 2.3, the Naming Service capability was not implemented. Instead, the Orbix-specific method was used to bind the client to the server, with the object reference returned to the client in the IIOP format. This method was much simpler to implement and seemed sufficient given the limited scope of CORBA applications on HiPer-D. Experiments with the Naming Service should run in future CORBA development, to achieve CORBA compliance.

f. When running Orbix 2.3 applications, there are a few very important environment variables to specify as follows:

(1) IT_CONFIG_PATH – This specifies the location of the Orbix configuration file (Orbix.cfg) which contains initialization for the main Orbix environment variables. To use a customized configuration file, this variable should point to location of user's personal Orbix.cfg.

(2) IT_ERRORS – This variable specifies the file containing the standard Orbix error messages. This is very useful during debugging.

(3) IT_DAEMON_PORT – This variable specifies the port on which Orbixd listens for requests. This variable is important because multiple users can run without interfering with one another, by choosing unique variable values other than the default value (1570).

(4) LD_LIBRARY_PATH – Include the Orbix lib directory here; otherwise there is no CORBA functionality.

4.3 Engagement Server Lessons Learned

The Engagement Server design depended on several features of the Isis middleware tool. A goal in the Engagement Server was to identify the minimal set of Isis features required to build a fault tolerant engagement service. This was especially pertinent in light of the ensuing Isis

obsolescence and it's subsequent replacement by another middleware tool. These features are enumerated here because they are relevant to the discussion of lessons learned.

(1) FIFO message delivery between a transmitter and a receiver.

(2) Reliable and Atomic delivery of messages transmitted to a process group (i.e. all surviving members or no members receive the message).

(3) Group membership events are ordered with respect to the message flow in the group.

4.3.1 Synchronization and Determinism

a. One of the design goals of the Fault Tolerant Engagement Server was to minimize hand shaking among the primary and shadow replicas. Another goal was to take advantage of the parallelism inherent in a distributed computing environment. These goals helped shape the design of the Engagement Server. The servers use a semi-active form of a primary/shadow execution and recovery model. It is semi-active in that the shadow replicas are actually processing requests and not acting as complete shadows to the primary replica. It is a primary/shadow model in that the shadow replicas must wait for synchronization events originating from the primary.

b. A key synchronization event is a message from the primary replica that informs the shadow replicas which message to begin processing. This is done to ensure that all replicas are processing the input messages in the same order. This is necessary to ensure that internal state is maintained and updated consistently among the replicas. Engagement Servers receive messages in several groups as well as from several transmitters within each group. This collection of messages might not be received at all replicas in the same order. It is the responsibility of the primary replica to inform the shadow replicas which of these inputs to begin processing, thus creating the necessary level of ordered processing.

c. Based on the Isis features above, the primary replica can know that the shadow replicas have received, or will receive, the message it is informing them to process. It also can know that the shadow replicas will see messages from any transmitter in the same order that it is seeing them. (This is FIFO on a node-by-node basis, not a total ordering of all messages received at all nodes.) The primary replica can, therefore, know that the shadow replicas will begin to process this particular message from the specified node and be assured it is the same message.

d. At this point the shadow replicas are released to process the message at best possible speed. Only at points in their processing where non-deterministic activities occur, e.g. a timer expiration, do they need to wait for another synchronization message from the primary. When the shadow replicas complete the processing for this message they begin performing another aspect of synchronization with the primary replica. Each shadow replica must verify all of the resultant transmissions from the primary replica. Each shadow replica receives these resultants in the appropriate groups and can immediately attempt to match and remove them. This will be the same set of resultants that each shadow replica has determined must be transmitted. This has not added any extra communications overhead because the middleware is already performing atomic multicasts to all group members.

e. The question this raises is "How can we be assured that each replica has come to the same conclusion about the processing for any given message?" Also, "How can we be sure each replica has determined the same set of resultants?" As previously mentioned, all replicas proceed in processing a message at best possible speed once the primary replica has notified all shadow replicas which message to process. Due to the nature of Engagement Server algorithms the only place where comparisons, calculations, or actions could produce differing results is in the timer expiration for an engagement response. Errors could arise, though, from the host and network environment. In a language such as Ada, hardware faults, out-of-range values, data corruption, etc. will manifest as exceptions. These exceptions are propagated into the application code for the replicas to handle. Error recovery can commence in these exception handlers. This error processing requires handshaking among the replicas to determine the appropriate course of action. Time did not permit the completion of this aspect of the design; therefore, it is still in development. But, apart from such errors, the replicas will agree on the processing conclusions and the set of resultants. Non-determinism only enters in at the point of timer expiration. An indicator from the primary replica handles this by specifying whether the shadow replicas should proceed with normal processing for this engagement request or enter the time-out processing.

f. An additional point here in regard to determinism is that the Engagement Server design is not sensitive to process or thread suspension. Suspensions of this type will not have a " value" effect on the engagement state data. Again, this is because of the nature of the state data that is maintained by the replicas. Timer expirations could be affected by such suspensions, but a synchronization indicator from the primary replica already handles this. This reveals that completely deterministic execution is not necessary for fault tolerance. Determinism as it relates to the state data maintained by the replicas is what is important and necessary.

g. There are some weaknesses with the described approach. First, applications that perform heavy mathematical and floating-point calculations would require additional handshaking. Different microcode precision in each processor could produce different or deviating results in the redundant copies. But, additional handshaking for value comparison would address this issue. Second, a robust error detection and exception handling approach must be taken in the application code. This is not difficult to do in Ada but must be done systematically and in layered fashion. Third, some knowledge of what constitutes a non-deterministic activity relative to the state data being maintained must be built into the application code. This requires some savvy on the part of application designer and coders. This is specifically true in this approach, but is true of fault tolerant programming in general. Finally, this approach does not address N-Version programming.

h. There are strengths of the approach as well. First, it is a very efficient performer in the tactical realm. The overhead to the primary replica is negligible. It has a couple of synchronization messages it must transmit to the shadow replicas. The shadow replicas have slightly more overhead because they must perform verification of the resultant transmissions. In fact, that is the majority of their overhead. Second, the handshaking has been minimized to reduce network traffic. The only additional messages generated by this approach are the synchronization messages. Third, the overhead cost is high only when it needs to be. The design is efficient in handling process failure. It is when errors arise that the overhead and

synchronization costs increase. But this is where they must increase because of the high probability of state data inconsistency. The probability of these error events is low; therefore, the design only pays this high synchronization cost a low percentage of the time. This approach has used minimal handshaking as well as capitalized on parallelism in the distributed environment. With those design attributes it has constructed a synchronized and deterministic fault tolerant engagement service.

4.3.2 Cross-Group Data Difficulties

a. As mentioned in the previous section, Engagement Server replicas join several groups to send and receive messages. In the case of group membership change events, Isis guarantees that these events are ordered in the message flow for that group. In other words, all group members will see notification of new or lost members at the same point relative to the message flow. This attribute is provided on a group by group basis. Since the replicas join several groups a failing replica will generate several of these events, one for each group it fails out of. The surviving replicas will see each of these events in the respective groups. The difficulty arises in that there is no ordering guaranteed across the groups because each change is a group event independent of other groups.

b. The Engagement Servers receive ID information about a track in an ID group, engagement requests in an Engagement group, etc. The engagement state data is built by combining and processing these pieces of data from different groups into a composite structure. When failure of the primary replica occurs, the shadow replicas will detect these membership change events. The shadow replicas must synchronize these events across all of the groups. This is done to ensure that each replica has the proper view of state held by the primary replica when it failed. This is a non-trivial task, but one that was accomplished by the suspension and resumption of message queues internal to each of the shadow replicas. The shadow replicas suspend internal queues as they detect the loss of the primary replica. The shadow replicas then resume these queues upon receipt of the primary indication from the shadow replica that has assumed the primary role. This allows the shadow replicas to guarantee a consistent view of the failed primary replica's state prior to its failure.

c. Ideally, systems could be built such that this intersection of data across groups would not occur in any component. Then components would not have to perform this cross group synchronization. This is not a realistic assumption in our advanced computing systems. While some components could be constructed in this manner, it will be difficult if not impossible to build all components such that their state was constructed from data sent and received in only one group. In any component where this is not the case, cross group synchronization will be a component-level task unless there are middleware tools that provide a synchronization capability across a user-specified set of groups. This synchronization capability currently exists in the Engagement Server design and it is a HiPer-D work-in-progress to modify this into a layered, adaptable capability.

4.3.3 Recovery Time and Group Coupling

a. There were two types of application changes to the fault tolerant Engagement Server that impacted the group structure. Some legacy functionality was moved from the Engagement Server to a more appropriate HiPer-D component. There were also some minor changes to the data formats of messages transmitted in existing groups. Both of these types of changes allowed the Engagement Servers to join fewer overall groups. These group reductions improved the overall timeliness of failure detection and recovery in the Engagement Server replicas. The specific recovery time was apparent when examining the Auto_Special Timeline display during Demo98 (refer to Section 3.7.3.2, Figure 3.7.3.2-3 for a discussion and example of this display). During normal SPY-Declared Auto_Special engagements the total processing required by the HiPer-D components took only 10-20% of the allowable AEGIS requirement. When the primary Engagement Server replica was intentionally failed, during such an engagement request, the percentages grew to 140-200%. This difference of 130-180% is the group-based recovery time required by the Engagement Servers. This recovery time includes the failure detection, notification, and synchronization provided by the middleware as well as necessary processing within the surviving replicas. One of the shadow replicas must become the new primary replica and complete the processing steps associated with the current engagement request. While, the recovery time indicated here is within AEGIS requirements, it was not necessarily bounded by the Isis middleware used in Demo 98.

b. The timeliness of recovery is greatly impacted by the level of "group coupling" in the system. In the group-based programming model all components that join common groups are "coupled" together by virtue of these common groups. This coupling is a necessary attribute of group-based middleware that allows it to provide message and event orderings as well as delivery guarantees. Upon failure of a group member all surviving members are notified of this membership change event. This event is ordered in that all surviving members will receive it at the same point relative to message traffic occurring in the group. The middleware is performing a hand shaking service to provide this ordered view of the failure event. In the middleware all of the surviving members must agree to the new view of membership, i.e. minus the failed member, before they can pass this event to the application. If any surviving member is slow to come into agreement then all members will be delayed until consensus is reached. This slowness will manifest itself as longer recovery times because of the increased latency between the actual failure and the application notification of such failure.

c. There could be many reasons for application slowness with respect to a new membership view. For example, it could be executing on a badly overloaded host or be suffering from priority inversion and, in both cases, be delayed in coming to consensus. There are architectural issues as well. The group coupling must be carefully analyzed and engineered. The recovery times will be lengthened if groups are very large because many members must come into consensus for each membership change event. The data required by each application component, what group(s) that data must be transmitted in, etc., are critical decisions in eliminating unnecessary group coupling. In the Engagement Server design, effort was made to reduce coupling as much as possible. Obviously, the existing HiPer-D group architecture had an impact on the reduction level that could be achieved. The critical issue is that group coupling

will have a direct impact on failure detection and recovery timeliness. These group architecture decisions must be considered at the beginning of the Engineering Process where that is possible.

4.3.4 Precise Fault Injection

As was shown in Demo 98 the Engagement Servers were able to recover from a fault that was precisely injected into the primary replica. This fault injection capability was described in Section 3.5.4.3. This technique made an important contribution to the testability aspects of the system. Our Advanced Computing Systems will contain hundreds of processes executing of hundreds of platforms. The ability to test the system's behavior in response to various faults cannot be overstated. There must exist mechanisms to inject faults of various types into the system and assess the system's response both functionally and with respect to performance. This was shown in Demo 98 by the ability to directly assess the impact of a primary replica failure during a Spy declared Auto_Special engagement. This analysis revealed an increase in the time required to process that engagement request; an increase due to the recovery time of the shadow replica that was assuming the primary role. But, the ability to perform this analysis is solely due to the precise fault injection capability developed in the Engagement Server work. General-purpose tools of this nature are not yet forthcoming, but they are essential to overall system validation.

4.4 Remote Digital Call for Fire (CFF) Lessons Learned

a. It was shown successfully that a remote FO/FAC spotter is capable of issuing digital CFF request and updates using the interfaces and equipment in the Advanced Computing testbed. The demonstrated functionality is considered representative of a potential real-world sequence that can serve as a basis for follow-on experimentation.

b. Improvements in track insertion processing depicting the CFF target were addressed, and the Demo 98 design is considered to be a more realistic design than that used in previous demonstrations. As a result, new issues relative to dependencies on the OTH track path have been identified for pursuit in out-year efforts. In particular, the value of creating a CFF dependency on the timely and reliable reception of an OTH track by the combat system needs to be revisited.

c. The need to support multiple CFF targets simultaneously was identified. Due to program constraints, the implementation was confined to a single engagement in Demo 98.

d. From a logistics standpoint of executing a test or demonstration, having a remote operator in a separate lab was, at best, an awkward and problematic configuration. There was never a guarantee that the actual network path would be available between the two locations. Invariably, periodic difficulties arose during process start-ups to establish network connections between the two labs. Furthermore, from the standpoint of smoothly staging a CFF engagement into the flow of an overall demonstration, a telephone connection between the buildings was required. Voice communications between the two labs during the course of a demo added additional coordination overhead and required the presence of an operator for extended periods of time in the remote building.

e. Coordinate conversion and accuracy issues were only addressed at a coarse-grain level. Further analysis and experimentation are deemed necessary in this area.

f. Engagement deconfliction was only assessed to the degree where an operator could perform this task visually. Further concept definition and system engineering of this function are also called for.

h. Future work is warranted to move towards variable format Tacfire messages as well as insertion of the latest NSFS prototypes and simulators in order to keep up with this rapidly moving area. Nevertheless, HiPer-D now has a respectable prototype approximation of this future capability.

4.5 RM Lessons Learned and Future Direction

a. Demo98 has demonstrated significant Resource Management capabilities that involve monitoring, decision-making, control, system specifications, and visualization. Figure 3.4-2 in Section 3.4 shows the high-level view of the data flow that occurs in this type of architecture.

b. Accomplished goals for Demo98 include:

- General-purpose dynamic resource allocation
- Application scale-up based on load and QoS assessment
- Reconfiguration based on fault-recovery, damage-status, damage-prediction, and recognition of QoS problems
- Dynamic monitoring and instrumentation across three UNIX platforms
- Human operator and automated control interface across three UNIX platforms
- Startup and shutdown control of third-party applications (ATWCS) as well as resource management infrastructure components
- Initial host allocation for selected AEC applications
- QoS performance analysis at application and path levels
- Host selection algorithms based on improved load metrics
- Monitoring, decision-making, and control driven by System Specification files
- Integrated visualization tools

4.5.1 Monitoring

a. Runtime monitoring with data-collection and storage are key to understanding the "health" of a system of resources. Providing monitoring services for each resource can be difficult. For instance, a compute-intensive daemon collecting massive amounts of data, or simply a tight performance requirement on a monitoring daemon will cause excess resource utilization and begin to defeat the purpose of providing resource management. An example was seen with SNMP daemons/services provided through the operating system on a DEC Alpha. In order to collect the data necessary for performance evaluation as well as process status, the workstation would use upwards to 80% of its resources executing SNMP requests. Obviously

this is an unacceptable solution to monitoring. Hence, special care must be taken to ensure that the monitoring components do not place a significant load on a computer's resources.

b. Since it is critical to provide monitoring daemons on all platforms in the resource pool, it therefore becomes important to provide monitoring support for multiple platforms. Our current monitoring components run on UNIX platforms. Future endeavors will include porting of these components to other operating systems such as Windows NT, Linux, VxWorks, and LynxOS.

c. Demo98 encompassed thirty-four UNIX workstations. In future years, the number of host platforms included in the demonstrations are expected to increase significantly. Supporting our goal for a diverse, configurable, upgradeable resource pool translates to requirements for a highly scalable monitoring infrastructure.

4.5.2 Resource Management Decision Making

a. The decision-making portion of Demo98 consisted of three major components: QoS Path-Managers, Resource QoS Managers, and Resource Manager. The inter-process communication for the Decision Making components was provided by a package developed at the University of Texas at Arlington (UTA) which was built on TCP/IP and required a name server for location independence. The communications package performed well but had several drawbacks. The name server was a bottleneck for scalability and survivability. In addition, the communications library performed dynamic memory allocation during message sends and receives which resulted in less than optimal performance. As a result, the choice of communication mechanisms being used within the Resource Management decision making components will be reassessed.

b. The Resource QoS Manager proved to be much more effective than the previously used FY97 "voting" mechanism. Resource allocation times were reduced from 10-12 seconds to under 500 milliseconds. The biggest difference involved precalculating host "fitness" scores for all hosts in the resource pool with periodic updates being performed every second to maintain up-to-date data. In addition to these changes, we also need to "normalize" the performance of each host. This would allow a symmetric multi-processor server to have a higher score than a single processor desktop or a Sun Ultra-60 to be "better fit" than a Sun Sparc-10.

c. The Resource-Manager component in Demo98 performed initial allocation, reallocation, and scale-up based on overload detection. Additional features will include scale-down based on underload, application move to correct poor performance, and pro-active scaleup based on prediction of overload. Application profiling, both in the static and dynamic sense, will also be an important key enabler for better resource allocation decisions. Static profiling will allow benchmarking to occur offline in a closed system and data provided to the Resource-Manager at runtime. Dynamic profiling will allow data to be collected and assessed at runtime.

4.5.3 Resource Control / Program Control

a. Program Control was the key component for providing startup and shutdown capabilities both by an operator and automated by the Resource Manager. A user-friendly display allowed operators to easily take control of tens to hundreds of hosts in a distributed, heterogeneous environment.

b. One issue that arose was the role of the display and control processes. During the course of the design and development of the Program Control components, the line between control and operator interface capabilities became blurred with the display handling several features that probably should have been implemented within the control process.

c. Security is another significant issue with distributed control. With the power to startup and shutdown processes across a large resource pool brings the problem of who is privileged to start and stop processes and on which machines. User-level control will need to be incorporated into the display and control daemons to ensure that only process owners and privileged users can control the system configuration (e.g., stop running processes, etc...).

4.5.4 System and Software Specifications

a. The Resource Management System and Software Specifications Grammar, RMSpec, was introduced in Demo98. It provided a convenient and centralized way for requirements for QoS, configuration, dependency, etc., to be delivered to the resource management components. Application software changes and system requirements changes that affect system performance are handled by simply modifying a specification file. Therefore, a 65ms Auto-Special review time can easily be lowered to 50ms.

b. Several implicit assumptions of the specifications grammar are concerns with regards to the general-purpose nature of the grammar. For instance, paths are defined with implicit concepts of "load". A more generic, explicit approach must be pursued for the RMSpec grammar. There needs to also be a way to "tag" data coming from the monitoring components and correlate it for use in the grammar. This way a path can be defined and measured by using text strings that map to data being delivered through instrumentation.

4.5.5 Visualization

a. Visualization is a key component for presenting data and decisions to operators. A glimpse of the Host Display quickly shows which processes are allocated to which processors, the status of all hosts, and the status of the network connections on all the hosts. The Graph Display allowed hardware performance (i.e. Cpu-Idle, Network Packets-In and Out, Memory Paging, etc...) to be monitored at runtime, thus keeping the operator informed as to the "health" of the hardware systems in the resource pool. The Path Display graphically depicted the QoS paths identified in the specifications files and allowed runtime plots of the data instrumented along those paths. Finally, the Resource Management Decision Display informed operators of all actions performed by the Resource-Manager including insight into the reasoning behind the specific allocations.

b. In order to monitor QoS requirements placed on applications and systems in Demo98, a modified version of the Jewel Instrumentation package was used. The displays used were very similar to those in previous demonstrations. A more generic approach to instrumentation displays is planned which would allow a display to be built using drag-and-drop techniques and hooked to data through a point-and-click mechanism built into the display and instrumentation infrastructure.

4.5.6 Summary

Demo98 unveiled a sound infrastructure base for dynamic resource management. Further algorithm development to utilize more of the available data will allow for better and more robust allocation and reallocation decisions to be made. Evolution of the existing architecture and the features that lie within will move us closer to our goal of providing a scalable, fault-tolerant, generic, dynamic resource allocation and control infrastructure.

APPENDIX A

EVALUATING THE PERFORMANCE OF MULTICAST COMMUNICATIONS

Philip M. Irey IV and David T. Marlow
System Research and Technology Department
Combat Systems Branch
Naval Surface Warfare Center, Dahlgren Division
Dahlgren, Virginia 22448-5000
{pirey, dmarlow}@nswc.navy.mil

APPENDIX A

EVALUATING THE PERFORMANCE OF MULTICAST COMMUNICATIONS

Philip M. Irey IV and David T. Marlow
System Research and Technology Department
Combat Systems Branch
Naval Surface Warfare Center, Dahlgren Division
Dahlgren, Virginia 22448-5000
{pirey, dmarlow}@nswc.navy.mil

ABSTRACT

In the distributed shipboard environment of interest to the United States Navy, there is an increasing interest in the use of multicast communications to reduce bandwidth consumption and to reduce latencies. The bandwidth required to transmit large volumes of information (e.g. track files, maps, etc.) to multiple receivers could potentially be reduced significantly by the use of multicast data transmission. Many types of real-time shipboard data, such as navigational and gyro data, need to be distributed to a large number of hosts. The distribution of this type of data might also benefit from the reduced latency possible using multicast techniques instead of sequential unicast transmission. Before multicast communications can be used in this environment, however, a characterization of its performance must be made. This appendix proposes a number of metrics, and data collection and analysis techniques for assessing multicast communications performance. Of particular significance is a metric that correlates reception of message and shows promise in analyzing topology-related problems. While the concepts presented in this appendix are applicable to the general forms of multicast, this appendix specifically focuses on the use of IP Multicast in an internal shipboard environment. The MCAST Tool Suite (MTS), which uses the metrics and data collection techniques presented, is then described. The results of applying this toolset to simulate and instrument several IP Multicast-based application scenarios is then presented.

A.1 INTRODUCTION

a. In the distributed shipboard environment, there are many data streams that multiple hosts need to receive. These data streams may be large in volume and sent often (in some cases, hundreds of times per second). Examples of such data streams include gyro and other positional update information. Multicast transfer, where each message forwarded by a sender is passed on to multiple receivers, is expected to decrease bandwidth resources on the interconnected data networks. In addition, it is expected that there will be reduced latency to pass each message to all receivers for this technique when compared to sequential unicast transmission to each receiver. The reduced bandwidth load is expected to enable other communications using the same network resources to gain bandwidth and low latency advantages.

b. The Internet Engineering Task Force (IETF) has defined IP Multicast [RFC1112] which provides the most universal approach to implementing multicast transfer with Commercial Off The Shelf (COTS) products. IP Multicast is an extension to the standard Internet Protocol

suite which provides mechanisms for receivers to register for particular message types and for routers to discover which multicast messages are needed by hosts it serves. IP Multicast provides a means of supporting multicast transfer to a number of hosts which may be interconnected by a variety of LAN types (e.g., Ethernet, FDDI or ATM). IP Multicast provides a connectionless (i.e., unacknowledged) transfer service on top of which reliable transfer mechanisms are being developed by the research community.

c. This appendix focuses on the use of UDP on top of IP Multicast which provides a connectionless transfer service for application software. It is assumed that a multicast routing protocol is in place among the routers or switches serving the multicast users. The details of these protocols are outside the scope of this appendix.

A.2 RELATED WORK

A large body of work has been published on unicast metrics [IREY98] and [IPPM]. Previous work in unicast metrics can provide a foundation for multicast metrics; however, the multicast transfer breaks some of the fundamental assumptions of unicast data transfer. Some of the previously defined unicast metrics can be used as a foundation and upon which new multicast metrics can be defined. The IETF has also initiated efforts in defining metrics targeted at multicast exchanges [BMWG].

A.3 MULTICAST MODEL

a. This appendix focuses on $1 \rightarrow N$ communications model as shown in Figure A-1.

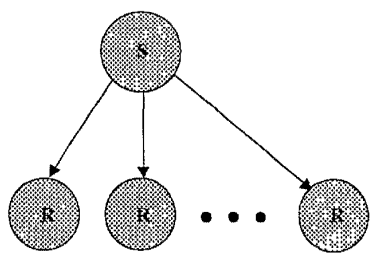


Figure A-1 1-N Communications Model

b. While, $1 \rightarrow N$ groups can be constructed from a series of unicast communications, the metrics for measuring the performance of these groups are outside the scope of this appendix and will not be discussed.

A.4 MULTICAST PERFORMANCE METRICS

Although this appendix specifically focuses on IP multicast, the metrics defined in this section have applicability to multicasting in general (e.g. 802.3 datagrams, ATM UNI point-to-multipoint connections, etc.). Two types of metrics are defined for accessing multicast performance: local metrics and group metrics. Local metrics are measured at a single sender or

single receiver. Group metrics, on the other hand, represent an aggregate performance for all receivers in a multicast group.

A.4.1 Metric Notation

Metric names are defined with capital letters (e.g. *XYZ*). An individual measurement of a metric from a set of measurements is represented using subscript notation (e.g. XYZ_i). The subscript $_i$ is used to select a particular measurement from a set of measurements. The subscript $_j$ is used to select a particular host from a set of hosts. To represent a particular measurement on a particular host, a double subscript notation is used. For example, $LIAT_{j_i}$ represents *LIAT* measurement number $_i$ on host number *j*. Statistics can also be applied to a set of measurements. Subscripts are used to represent the statistics computed for a set of measurements: "avg" represents the average value; "min" represents the minimum value; "max" represents the maximum value; and "sdev" represents the standard deviation. For example, XYZ_{avg} represents the average value of the set of measurements of the *XYZ* metric. To denote a statistical measurement from a particular host in group metrics, the statistic subscript again is subscripted. For example, XYZ_{avg_j} represents the average value of the *XYZ* metric on receiver *j*. In the equations below, *m* represents the number of messages sent on a data stream and *N* represents the number of multicast receivers in the group to which the data stream was sent.

A.4.2 Metric Instrumentation Points

Both the multicast sender and multicast receivers are instrumented at the application to communications subsystem interface as shown in Figure A-2. The communications subsystem is generally contained within the operating system (e.g. in an Unix environment, the application to communications subsystem interface is often the sockets API). Because the measurements are made at this interface, the performance observed by the application is measured and includes delays introduced by transmission on the media, queuing delays in the communications subsystem, and delays in the operating system.

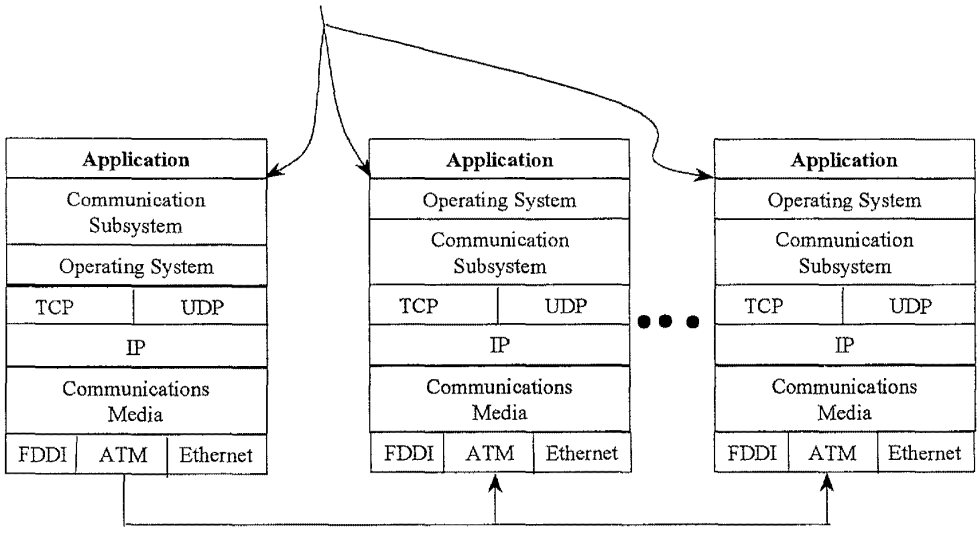


Figure A-2 Metric Instrumentation Points

A.4.3 Messages

a. Messages are units of data sent from the multicast transmitter to the multicast receivers in the group. They are application level entities as opposed to packets which may be used within the communication subsystem. Performance data is collected on individual messages or a stream of messages.

b. Each message sent contains a sequence number, (seq), a timestamp, (ts1), and data. The sequence number is used by multicast receivers to order messages and to determine the distance (in messages) between messages in the data stream. In this appendix, sequence numbers are assumed to be monotonically increasing starting from one on a data stream from a particular transmitter (e.g. increased by one each time a message is transmitted). Other sequence numbering algorithms can be used as long as they allow message order to be determined and the distance between messages to be computed. The timestamp identifies when the message was sent by the transmitter and is used by the receivers to measure latency and throughput. Finally the data is "filler" at the end of a message to ensure that the message is a particular size. The size of a message is important when trying to simulate a particular application scenario.

A.4.4 Local Metrics

a. Metrics with the "local" prefix are measured at either the single sender or at a single receiver as shown in Figure A-3.

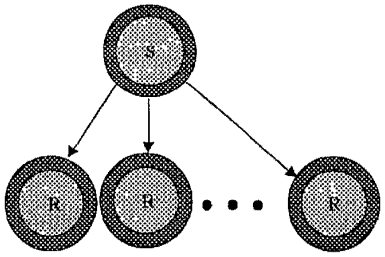


Figure A-3 Local Metric Instrumentation Points

b. The "local inter-send time" metric and "local messages sent" metrics are measured at the sender. All other "local" metrics are measured at a single receiver. Even though these metrics are measured at a single receiver, information from the sender (e.g. sequence numbers, timestamps, etc.) may be used to compute them. Figure A-4 and A-5 show pseudo-code similar to the "C" programming language which shows the operation of the sender and receiver. It should be noted that the local metrics are appropriate for the measurement of local multicast performance and for local unicast performance as well. Unicast transfer is essentially an exchange between a single sender and a single receiver. In this case, the metrics are valid regardless of whether a unicast protocol (e.g. UDP over IP) or a multicast protocol (e.g. UDP

over IP multicast) is used to exchange the data (though this may affect the measured performance).

```
seq=LMS=0
while (done==FALSE) {
        ts1=gettimeofday(ts1)
        send(A,++ seq,ts1,data)
        ts2=gettimeofday(ts2)
        LIST[ ++LMS]=timediff(ts2,ts1)
        local_usleep(sleep_time)
}
```

Figure A-4 Transmitter Pseudo-Code

c. As shown in Figure A-4, messages are sent from the multicast transmitter to the receivers using a send() function. Each message is sent to the address A which can address a unicast receiver or a multicast group of receivers. Each message sent contains a sequence number, seq, a timestamp, ts1, and data. Since the timestamp is generated at the transmitter and is used by the receiver, it is assumed that either the clocks of the sender and all receivers are synchronized (e.g. NTP, GPS, etc.) or some method exists for converting between the time domains of the transmitter and receivers [NTP].

d. As shown in Figure A-5, multicast receivers use the recv() function to receive the messages sent to the address A and extract the sequence number, seq, timestamp, ts1, and data from those messages.

A.4.4.1 Local Messages Sent (LMS) Metric

As shown in Figure A-4, *LMS* is computed at the transmitter. It is simply the count of the number of messages sent to the group.

```
bytes_received=LMR=LMRL=LGC=0
expected_seq_num=0
tr1=gettimeofday()
first=TRUE
while (done==FALSE) {
        prev_tr1=tr1
        recv( A, seq,ts1,data)
        tr1=gettimeofday()
        if (first == TRUE) {
                start_time=ts1
                FIRST=FALSE
        |
        process_packet = FALSE
        if (seq == expected_seq_num) {
                process_packet=TRUE
        }
        else if (seq < expected_seq_num) {
                ++ LMRL
        }
        else { /* seq > expected_seq_num */
                LGB[ ++LGC]= (expected_seq_num,seq)
                LGL[LGC]=seq-expected_seq
                process_packet=TRUE
        }
        if (process_packet) {
                LOWL[ ++LMR]=timediff(tr1,ts1)
                LIAT[LMR]=timediff(tr1,prev_tr1)
                expected_seq_num=seq+1
        }
        bytes_received=bytes_received+
            sizeof(data)
}
end_time=gettimeofday()
LAT=bytes_received/timediff(end_time,start_time)
```

Figure A-5 Receiver Pseudo-Code

A.4.4.2 Local Inter-Send Time (LIST) Metrics

The *LIST* metrics are used to statistically characterize the time between successive message sends at the multicast transmitter as shown in Figure A-4. Computing *LIST* is important because the measured value may not always correspond to the expected value as shown in [IREY97]. The "sleep_time" or target time between successive message sends directly affects the *LIST* values measured. Setting sleep_time is useful in simulating specific application scenarios. The average, maximum, and minimum values can be computed for a set of *LIST* measurements as shown in Equations 1, 2, and 3 respectively. The standard deviation for a set of *LIST* measurements can be computed as shown in Equation 4.

$$LIST_{avg} = \sum_{i=1}^{m} \frac{LIST_i}{m} \qquad \text{(EQ 1)}$$

$$IST_{max} = \underset{\forall i}{MAX}\{LIST \qquad \text{(EQ 2)}$$

$$LIST_{min} = \underset{\forall i}{MIN}\{LIST \qquad \text{(EQ 3)}$$

$$LIST_{sdev} = \sqrt{\frac{\sum_{i=1}^{m} (LIST_i - LIST_{avg})^2}{n-1}} \qquad \text{(EQ 4)}$$

A.4.4.3 Local Messages Received (LMR) Metric

As shown in Figure A-5, LMR_j is computed at a single receiver. It is the count of the number of messages received. LMR_j includes messages received in sequence and those whose sequence number was greater than the sequence number expected for the next message in the data stream. Messages received with a sequence number less than the expected one are not counted in LMR_j but are counted in *LMRL* defined below.

A.4.4.4 Local Messages Received In-order (LMRI) Metric

$LMRI_j$ is defined to count only the messages which were received with a sequence number equal to the expected sequence number as shown in Equation 5 (where LGN is defined in paragraph A.4.4.11).

$$MRI_j = LMR_j - LGN_j \qquad \text{(EQ 5)}$$

A.4.4.5 Local Percent Messages Received (LPMR) Metric

LPMR$_j$ is defined to compute the percentage of the messages sent by the multicast transmitter which were received by a multicast receiver. It is simply the number of messages received by the receiver (*LMR*$_j$) divided by the number of messages sent by the transmitter (*LMS*) as shown in Equation 6.

$$LPMR_j = \frac{LMR_j}{LMS} \qquad \text{(EQ 6)}$$

A.4.4.6 Local Messages Received Late (LMRL) Metric

LMRL$_j$ is defined to count the messages received by a receiver with seq less than expected_seq as shown in Figure A-5. Both messages which arrive out of order and duplicate messages are classified as "late" using this criteria. Although "late" packets are not common in the shipboard environment of interest in this appendix, they can occur (particularly in failure scenarios) and must be considered. In a Wide Area Network (WAN), "late" packets are more common.

A.4.4.7 Local One-Way Latency (LOWL) Metrics

LOWL metrics are used to statistically characterize the one-way latency between the multicast transmitter and a single receiver. These metrics are computed using the difference of two timestamps as shown in Figure A-5. The first timestamp, (ts1), is generated by the multicast transmitter when the message is sent. The second timestamp is generated at the receiver, (tr1), when the packet is received. $LOWL_j$ is the difference between tr1 and ts1 (assuming the timestamps are in or can be converted to the same time domain). $LOWL_{avg_j}$, $LOWL_{max_j}$, $LOWL_{min_j}$, and $LOWL_{sdev_j}$ are computed as shown in Equations 7 through 10. Since LMR_j is used in the computations, messages which are late or lost do not contribute to the computations.

$$LOWL_{avg_j} = \sum_{i=1}^{LMR_j} \frac{LOWL_{j_i}}{LMR_j} \qquad \text{(EQ 7)}$$

$$LOWL_{max_j} = \underset{\forall i}{MAX}\{LOWL_{j_i}\} \qquad \text{(EQ 8)}$$

$$LOWL_{min_j} = \underset{\forall i}{MIN}\{LOWL_{j_i}\} \qquad \text{(EQ 9)}$$

$$LOWL_{sdev_j} = \sqrt{\frac{\sum_{i=1}^{LMR_j}(LOWL_i - LOWL_{avg_i})^2}{LMR_j - 1}} \qquad \text{(EQ 10)}$$

A.4.4.8 Local Inter-arrival Time (LIAT) Metrics

LIAT metrics are used to statistically characterize the time between the receipt of successive messages at a single receiver as shown in Figure A-5. Since *LIAT* is computed using timestamps from the local receiver only, no time domain conversions are necessary. Since $LIAT_0$ is always equal to zero, it is not used to compute $LIAT_{avg}$ as shown in Equation 11. $LIAT_{max}$, $LIAT_{min}$, and $LIAT_{sdev}$ can be computed for the set of *LIAT* measurements in a manner similar to Equations 8-10 except that *i* starts at 2 and the LMR_j term in Equation 10 should be replaced with the term LMR_j-*1*. Since LMR_j is used in the computations, messages which are late or lost do not contribute to the computations.

$$LIAT_{avg_j} = \sum_{i=2}^{LMR_j} \frac{LIAT_{j_i}}{LMR_j - 1} \qquad \text{(EQ 11)}$$

A.4.4.9 Local Application-to-Application Throughput (LAT) Metric

The LAT metric is used to characterize the end-to-end throughput between the multicast transmitter and a single receiver. The term "application-to-application" is used because the time interval over which the measurement is made includes the time from which the first message of the data stream was transmitted at the sender until the time the last message in the data stream was received by the receiver. As shown in Figure A-5, the start of the time interval begins with the receiver recording the value of ts1 in the first message received on the data stream in start_time. The receiver then records the time when the last message was received in end_time. LAT_j is the number of bytes received divided by the difference between these two times.

A.4.4.10 Local Gap Boundaries (LGB) Set

a. LGB_j is the set of ordered pairs of the start and end points of gaps in sequence space of received packets. LGB_j is a set from which metrics are derived for host *j*. The storage space required to record which messages were received in order and which were not can be reduced by using an *LGB* set.

b. When a message is received with a sequence number which is not equal to expected_seq (i.e., one greater than the highest previously received in-order message), one of two scenarios occurs: 1) the message has a sequence number greater than expected_seq in which case the ordered pair (expected_seq_num,seq-1) denoting the sequence space gap is recorded in LGB_j; or 2) the message has a sequence number less than expected_seq in which case the message is considered late and no action associated with LGB_j is taken. A new sequence space gap is not recorded for the second scenario because the packet was either a duplicate packet

(which is actually counted as a late packet as shown below) or was covered by a previously recorded sequence space gap.

A.4.4.11 Local Gap Number (LGN) Metric

The *LGN* metric is a count of the sequence space gaps observed during an experiment. It is equal to the number of messages received which had a sequence number greater than expected_seq.

A.4.4.12 Local Gap Length (LGL) Metrics

The *LGL* metrics are used to statistically characterize the size of sequence space gaps observed by a single multicast receiver. LGL_{j_k} for sequence space gap LGB_{j_k} is computed by subtracting the ending sequence number from the starting sequence number as shown in Equation 12. (Figure A-5 shows (start, end)-tuples being recorded for LGB). LGL_{avg_j} is computed by iterating over k as shown in Equation 13. LGL_{max_j}, LGL_{min_j} and LGL_{sdev_j} are computed similarly to Equations 2 through 4, except that they are computed over k as in Equation 13.

$$GL_{j_k} = LGB(end)_{j_k} - LGB(start)_{j_k} \qquad \text{(EQ 12)}$$

$$LGL_{avg_j} = \sum_{k=1}^{LGC_j} \frac{LGL_{j_k}}{LGC_j} \qquad \text{(EQ 13)}$$

A.4.5 Group Multicast Metrics

a. Unlike local metrics which look at the performance of a single sender or a single receiver at a time, group metrics attempt to characterize the performance of all receivers in a multicast group as shown in Figure A-6.

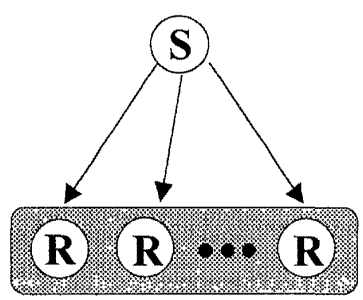


Figure A-6 Group Metric Instrumentation Points

b. Group metrics are only defined for multicast receivers. There are no group metrics for the multicast transmitter.

A.4.5.1 Group One-way Latency (GOWL) Metrics

GOWL metrics are used to statistically characterize the one-way latency performance of the group. *LOWL* measurements from each multicast receiver are used in the computation. The average, maximum, minimum and standard deviation of a set of *GOWL* measurements can be computed as shown in Equations 14 through 17. These equations also serve as prototypes for computing average maximum, minimum, and standard deviation on sets for other group metrics defined below.

$$GOWL_{avg} = \sum_{j=1}^{n} \frac{LOWL_{avg_j}}{n} \qquad \text{(EQ 14)}$$

$$\mathrm{GOWL}_{\max} = \underset{\forall j}{\mathrm{MAX}}\{\mathrm{LOWL}_{\max_j}\} \qquad \text{(EQ 15)}$$

$$\mathrm{OWL}_{\min} = \underset{\forall j}{\mathrm{MIN}}\{\mathrm{LOWL}_{\min_j}\} \qquad \text{(EQ 16)}$$

$$GOWL_{sdev} = \sqrt{\frac{\sum_{j=1}^{n}(LOWL_{avg_j} - GOWL_{avg})^2}{n-1}} \qquad \text{(EQ 17)}$$

A.4.5.2 Group Inter-arrival Time (GIAT) Metrics

GIAT metrics are used to statistically characterize the message inter-arrival performance of the group. *LIAT* measurements from each multicast receiver are used in the computation. $GIAT_{avg}$, $GIAT_{max}$, $GIAT_{min}$, and $GIAT_{sdev}$ are computed in a manner similar to Equations 14-17.

A.4.5.3 Group Application-to-Application Throughput (GAT) Metrics

GAT metrics are used to statistically characterize the end-to-end throughput performance from the transmitter to the receivers of the group. *LAT* measurements from each multicast receiver are used in the computation. GAT_{avg}, GAT_{max}, GAT_{min}, and GAT_{sdev} are computed in a manner similar to Equations 14-17.

A.4.5.4 Group Gap Number (GGN) Metrics

GGN metrics are used to statistically characterize the number of message gaps observed by the group. LGC measurements from each multicast receiver are used in the computation. GGN_{avg}, GGN_{max}, GGN_{min}, and GGN_{sdev} are computed in a manner similar to Equations 14-17.

A.4.5.5 Group Gap Length (GGL) Metrics

GGL metrics are used to statistically characterize the size of message gaps observed by the group. *LGL* measurements from each multicast receiver are used in the computation. GGL_{avg}, GGL_{max}, GGL_{min}, and GGL_{sdev} are computed in a manner similar to Equations 14 through 17.

A.4.5.6 Group Reception Correlation (GRC) Metric

a. The *GRC* metric is used to measure the degree of correlation in the messages received by members of the group. To compute *GRC*, reception vectors are created for each group member.

b. A reception vector is a representation which records which messages on a data stream were received in sequence and which were not received in sequence or not received at all by a particular multicast receiver in the group. To compute the reception vector for multicast receiver j, $\vec{V_j}$, component i of $\vec{V_j}$ is set equal to one if the message with sequence number i was received in order and is set equal to zero otherwise. The information needed to compute the reception vectors is recorded in *LGB*. The number of components in a reception vector is always equal to m.

c. A set of reception S can then be grouped into a reception matrix, R, as shown in Equation 18. Each column j of R contains the reception vector for multicast receiver j. Each row i of R contains a Reception Report (RR) for all receivers for message i. The number of rows in R is always equal to m. The number of columns in R is equal to $|S|$ (the cardinality of S). The reception matrix in Equation 18 shows R constructed from S which contains the complete set of reception vectors (e.g. all multicast receivers in the group). R can be constructed for any subset, S, of the complete set of reception vectors as well.

$$R = \begin{bmatrix} V_{11} & V_{12} & \cdots & V_{1n} \\ V_{21} & V_{22} & \cdots & V_{2n} \\ V_{31} & V_{32} & \cdots & V_{3n} \\ \cdots & \cdots & \cdot & \\ V_{m1} & V_{m2} & \cdots & V_{mn} \end{bmatrix}$$

Reception Report For Message #1

Reception Vector for Host #2

(EQ 18)

d. *GRC* is computed on a reception matrix R as shown in Equation 21 which is derived from Equations 19 and 20. Equation 19 is used to compute the Message Reception Correlation (*MRC*) for message i. In this equation, the sum of R_{ij} is the number of multicast receivers which received message i ($R_{ij} = 1$) and n minus this sum is the number which did not ($R_{ij} = 0$). The absolute value of the difference of these two quantities is then divided by n (the number of multicast receivers) which yields a value between 0 and 1. This value represents the degree of correlation among the multicast receivers in receiving message i. A degree of correlation equal to 1 indicates that all receivers in the group whose reception vectors are contained in R received that message. A degree of correlation equal to 0 indicates that half of the receivers received the message and the other half did not. The sum of the *MRC* values is computed and divided by m

(the total number of messages which could have been received) to give the average degree of correlation for all messages, or *GRC*, as shown in Equation 20. The values of *GRC* range from 0 to 1.

$$MRC_i = \frac{\left|\left(\sum_{j=1}^{n} R_{ij}\right) - \left(n - \sum_{j=1}^{n} R_{ij}\right)\right|}{n} \qquad \text{(EQ 19)}$$

$$GRC = \sum_{i=1}^{m} \frac{MRC_i}{m} \qquad \text{(EQ 20)}$$

$$GRC = \frac{\sum_{i=1}^{m} \left| 2 \sum_{j=1}^{n} R_{ij} - n \right|}{n \times m} \qquad \text{(EQ 21)}$$

A.4.6 Impact of Unreliable Data Transmission Environment

The utility of the metrics defined in Sections A.4.4 and A.4.5 are dependent on the test environment. In the environment of interest, IP multicast data transmission is used which has unreliable semantics which can affect any measurements collected.

A.4.6.1 Correlation of Group Receivers

The *GRC* computed for the group provides a measure of the consistency of data reception among the set of multicast receivers. If there is a low degree of correlation among the group members, it is unlikely that the receivers are making their measurements on the same set of samples. As an extreme example, suppose Host 1 receives all messages with odd sequence numbers and Host 2 receives all messages with even sequence numbers which were sent on the same datastream to a multicast group. In this case, Host 1 and Host 2 are making measurements on what can be viewed as two different sets of data. In this case, *GRC* is equal to zero. Non-gap group metrics should be viewed with caution in this case since *GRC* indicates no correlation among the receivers. The local metrics collected are still valid but care should be taken when comparing them among receivers. Cross checking of the data measurements is especially important in this case.

A.4.6.2 GRC Partitioning Algorithm

Hosts can be partitioned into groups based on the *GRC* metric computed for those groups. First, a threshold value is specified. Next, groups of hosts whose *GRC* is greater than or equal to the threshold value are created. Two partitioning algorithms are presented. The goal of *GRC* Loose Partitioning algorithm is to find the largest sets of hosts which meet the threshold criteria.

The goal of the *GRC* Strict Partitioning algorithm is to find the largest sets of hosts such that all subsets of each partition meet the threshold criteria.

A.4.6.2.1 Loose GRC Partitioning

A set Π is created to partition the set of receiving hosts into subsets based on their RR's such that the *GRC* for each subset is less than or equal to a threshold correlation, τ, as shown in Equations 22 and 23. The purpose of Equation 22 is to create the set *P* which contains all subsets of the set of RR's, *R*, which meet the threshold correlation. Equation 23 removes all subsets of *P* which are proper subsets of other elements of *P* to form the set Π which contains the largest subsets of *P* which meet the threshold correlation. Testing each of these subsets against the threshold criteria can be an expensive operation for large sets of RR's.

$$P = \{s \subseteq R | GRC(s) \geq \tau \qquad \text{(EQ 22)}$$

$$\Pi = \left\{ \begin{matrix} A \subseteq P | A \text{ is not a proper subset of} \\ \text{any other element of P} \end{matrix} \right\} \qquad \text{(EQ 23)}$$

A.4.6.2.2 Strict GRC Partitioning

A set Π is created to partition the set of receiving hosts into subsets based on their RR's such that the *GRC* for each subset and all subsets of that subset is less than or equal to a threshold correlation, τ, as shown in Equation 24 and 25. The purpose of Equation 24 is to create the set *P* which contains all subsets of the set of RR's, *R*, which meet the threshold correlation and all subsets of those sets also meet the threshold criteria. Equation 25 removes all subsets of *P* which are proper subsets of other elements of *P* to form the set Π which contains the largest subsets of *P* which meet the threshold correlation. It should be noted that there are 2^n subsets which can be generated from *R* (where $n = |R|$). Testing each of these subsets against the threshold criteria can be an expensive operation for large sets of RR's.

$$= \{s \subseteq R | \forall T, T \subseteq s, GRC(T) \geq \tau \qquad \text{(EQ 24)}$$

$$\Pi = \left\{ \begin{matrix} A \subseteq \ | A \text{ is not a proper subset of} \\ \text{any other element of P} \end{matrix} \right\} \qquad \text{(EQ 25)}$$

A.4.7 Cross-checking of Data Measurements

The metrics presented have been defined to operate in the presence of the loss, reordering, or duplication of data which can result if an unreliable multicast protocol is used (e.g. IP multicast). Despite this, unreliable transmission can result in what appears at first glance to be inconsistent results. Cross checking among the various measurements should be done. For

example, it is possible to measure a low value for *LOWL* and a high value for *LIAT* on a data stream. This seems inconsistent since the measured *LOWL* implies good network performance yet the *LIAT* implies poor network performance. In this case, examining *LGC* might show a large number of gaps which would contribute to the large value for *LIAT*. *LOWL* is still low since the time interval used for that metric only uses timestamps associated with that message. The *LIAT* metric, on the other hand, uses a timestamp taken when previous message was received on the data stream. If messages are lost, the measured value of *LIAT* becomes larger.

A.4.7.1 Test Termination

a. Another consideration when using an unreliable multicast protocol is how to identify the end of a data stream at the receiver. If messages can be lost, simply counting received messages at the receiver is not sufficient. Figure A-5 shows the receiver executes while the value of the variable "done" is equal to FALSE. Although the pseudo-code does not show the value of "done" being changed to TRUE, it is assumed that this will be done when the receiver has determined that the data stream is completed.

b. The algorithms for identifying the end of a data stream are not shown in the pseudo-code because it was felt that this is an implementation detail and does not affect the definition of the metric. A variety of methods exist for a receiver to determine when the data stream has terminated. MCAST, presented below, sends messages shorter than those in the data stream to terminate a test. Metrics affected by any delay introduced by the test termination algorithm should be adjusted appropriately. The only metrics affected by this delay are *LAT* and *GAT*.

A.5 THE MCAST TOOL SUITE (MTS)

a. MTS was developed at the Naval Surface Warfare Center - Dahlgren Division to enable the performance analysis of multicast systems. MTS performs data collection and analysis related to metrics defined in this appendix. MTS will be used in Section A.6 of this appendix for instrumentation and analysis

b. While the metrics are applicable to generalized multicast communications, the tools in MTS focus on testing the performance of UDP on top of IP Multicast. MTS consists of four types of components: an instrumentation tool, a test coordination tool, a data extraction tool, and several data analysis tools. Although these tools can be used individually, they are generally used together. Figure A-7 shows the relationships among the tools in MTS.

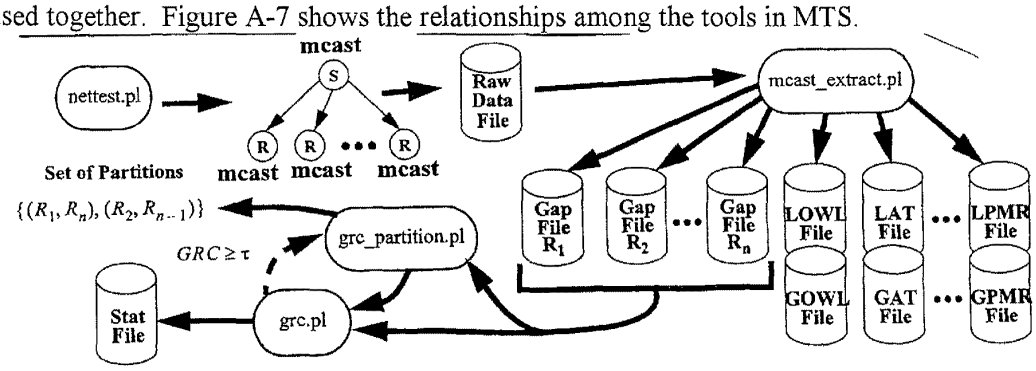


Figure A-7 MCAST Tool Suite

A.5.1 MTS Instrumentation Tool

a. The main tool in MTS is the Multicast Communications Analysis and Simulation Tool (MCAST). MCAST is a "C" program used to instrument a set of systems which are part of an IP multicast group. MCAST is run on the single sender associated with the group and on the receivers in that group which are to be instrumented. A multiple sender group is considered invalid for testing with MCAST.

b. The MCAST tool reports metrics on the sender and at each receiver (e.g. local metrics). Group metrics are not collected by MCAST, but are computed by the data extraction tools as shown in Section A.5.3.

c. The MCAST tool is controlled by command line options. To illustrate a simple test scenario, three command line options must be understood: "-s" denotes that this host is the multicast sender; "-r" denotes that this host is a multicast receiver; and "-i" specifies a specific network interface on which to send or receive. To start a test with a sender and two receivers, "mcast -s -i net_interface" would be run on the sending host, and "mcast -r" would be run on the two receiving hosts. Results of the test are presented on the sender and on each of the receiving hosts.

A.5.2 MTS Test Coordination Tool

a. For tests which involve a larger number of hosts than the simple case presented in the previous section, the process of starting the MCAST sender and receivers and gathering their results can be laborious. Each process must be started by hand with the appropriate command line options on the appropriate host. When the test is completed, the results reported by each of the processes must be collected from those hosts and then collated in a way that the group metrics can be computed. (MCAST only collects local multicast metrics). A tool called doit_multi.pl was developed to automate this process.

b. The doit_multi.pl tool interactively prompts the user to enter the names of the sending host and each of the receiving hosts. Next, the user is prompted to enter parameters associated with the test (e.g. Inter-send time, group address, etc.). In addition, doit_multi.pl allows the user to enter a range of message sizes over which to test instead of testing just a single message size as MCAST does.

c. After all of the test parameters are entered, doit_multi.pl starts MCAST with the specified parameters on the specified hosts and gathers the results from those hosts into a single output file where it can be processed by the data extraction tool presented in Section A.5.3. If a range of message sizes was selected, doit_multi.pl repeats the process for each of the message sizes tested, appending the results of each message size to the output file.

A.5.3 MTS Data Extraction Tool

The output file generated by doit_multi.pl can be quite large, particularly if testing is done over a range of message sizes. The data in this output file is in a raw format suitable for visual inspection but not for plotting or numerical analysis. To transform output files generated by doit_multi.pl into a form suitable for plotting or numerical analysis, a program called mcast_extract.pl was developed. This program extracts the data contained in a doit_multi.pl output file into three types of data files: gap files, local metric files, and group metric files. Output file names are generated by mcast_extract.pl and are based the name of the input file (which contains the output of doit_multi.pl). In the data file type descriptions below, the name of the input file passed to mcast_extract.pl is assumed to be "ifile".

A.5.3.1 Gap Files

a. Gap files describe the start and end of sequence gaps in the stream of messages received during a test. These files are a representation of the Local Gap Boundary (LGB) set described in Section A.4.4.10. Gap files have the following naming convention: "ifile.gaps.node". Here, "ifile" is the name of the input file passed to mcast_extract.pl, ".gaps." is generated by mcast_extract.pl to identify the type of file, and "node" is the name of the host corresponding to the data file. One file is generated for each host whose data is collected by the doit_multi.pl script.

b. The format of each line of a gap file is as follows: "msglen 0_or_1 start end". Here, "msglen" is the length of the message being tested by doit_multi.pl. (doit_multi.pl can test a over a range of message sizes during a single test.) "0_or_1" is either a zero which means all messages in the interval specified by "start" and "end" have not been received or a one which indicates that all messages in the interval were received. "start" indicates the sequence number of the start of the interval being reported and end indicates the sequence number of the end of that interval. The value of "0_or_1" should alternate between zero and 1 for a given value of "msglen" in a data file.

A.5.3.2 Local Metric Files

a. Local metric files contain metrics measured at each multicast receiver. These files contain a variety of metrics presented in Section A.4.4. Local metric files have the following naming convention: "ifile.local.metric". Here, "ifile" is the name of the input file passed to "mcast_extract.pl", ".local." is generated by mcast_extract.pl to identify the type of file, and "metric" is the type of metric collected in the data file. Current values for "metric" are: "at" for Application-to-Application throughput (see *LAT* defined in Section A.4.4.9); "iat" for Inter-Arrival Time (see *LIAT* defined in Section A.4.4.8); "owl" for One-Way Latency (see *LOWL* in Section A.4.4.7); and "pmr" for percent of the messages received (see *LPMR* in Section A.4.4.5). The complete set of local metrics defined in this appendix will be added to mcast_extract.pl in the future.

b. The format of a local metric file contains two parts: the key header section and the data section. The key header section provides a description of the data contained in each column

of the data section which follows. Each line of the key header has the following format "# column X = metric_ss host": where the "#" prefix is used to denote a comment in plotting programs such as gnuplot, "column X" indicates which column is being described by the key header entry, "metric_ss" is the search string used to extract the data from the raw doit_multi.pl input file for this column, and "host" is the name of the host for which the data in the column corresponds. As described in the key header section, the first column of each line of the data section contains the message length. Each column thereafter of each line in the data section contains the data collected for a particular host as described in the key header section.

c. The format of the local metric file is intended to allow plots to be easily generated with plotting tools such as gnuplot. In gnuplot, for example, column 2 versus column 1, column 3 versus column 1, column 4 versus column 1, and so on can be plotted simultaneously to show the local metric for each receiver on the same graph.

A.5.3.3 Group Metric Files

a. Group metric files contain metrics derived from the local metric files. The metrics correspond to those defined in Section A.4.5. Group metric files have the following naming convention: "ifile.group.metric". Here, the input file passed to "mcast_extract.pl", ".group." is generated by mcast_extract.pl to identify the type of file, and "metric" is the type of metric collected in the data file. Current values for "metric" are the same as those defined for local metric files in Section A.5.3.2. One group metric file is generated for each local metric file generated.

b. The format of group metric files is also the same as that of the local metric files (i.e. a key header section followed by a data section). These files are also in a format suitable for plotting. Group metric files always have two columns of data. The first column contains the message length. The second column contains the measured value of the group metrics.

A.5.4 MTS Data Analysis Tool

Once the data has been extracted into the various metric files, the data analysis tools available in MTS can process it. The tools enable analysis to be done related to the Group Reception Correlation defined in Section A.4.5.6.

A.5.4.1 GRC Processing Tool

A tool called grc.pl was developed to compute the *GRC* for a set of hosts. The *GRC* can be computed for an arbitrary number of hosts involved in a test. The list of hosts are specified on the command line. The tool outputs two columns of numbers. The first column contains the message length and the second column contains the *GRC* computed for the message lengths specified on the given hosts. Options available to grc.pl are: "-thresh" to specify a *GRC* threshold, "-stats" to output detailed statistics related to the *GRC* computation, and "-msglen" to compute the *GRC* on a single message length only. The threshold value is used to specify the *GRC* level required for all message lengths tested to be considered successful. The *GRC*

processing tool uses the return code of the program to indicate that either all *GRC* values computed for all message lengths tested meet the threshold value or at least one does not.

A.5.4.2 GRC Partitioning Tool

a. A tool called grc_partition.pl was developed to partition hosts into performance groups based on the a *GRC* threshold value as described in Section A.4.6.2. Because there are 2^n possible subsets which must be tested against the threshold criteria given n hosts, the complete enumeration of sets is too costly to compute. A Loose GRC Partitioning algorithm is used by the tool.

b. The algorithm used by the tool tests $0(n^3)$ sets rather than the 2^n sets required if a Strict GRC Partitioning is done (e.g., all sets are tested). The algorithm is guaranteed to produce one or more sets which meet the threshold criteria rather than the largest number of sets which meet the threshold criteria.

A.6 IP MULTICAST PERFORMANCE TESTS

A number of tests were conducted to evaluate the performance of UDP over IP multicast using the metrics developed in Section A.4. Some of the tests were performed in a "stand-alone" configuration outside the HiPer-D testbed. Other tests were "integrated" in the sense that they were conducted in the HiPer-D testbed to solve specific operational problems.

A.6.1 Stand-alone Testing

A number of experiments were conducted outside the HiPer-D testbed to prove the utility of the multicast performance metrics and MTS. Eventually the tools evaluated and applied during the stand-alone testing were transitioned into the HiPer-D testbed where they were used.

A.6.1.1 Stand-alone Test Environment

The test environment consists of a number of Sun Microsystems workstations and servers (Ultra 2, SparcStation 5, Ultra Enterprise 450, and SparcServer 4/ 670 MP) running Solaris 2.6. Tests were conducted over one Gigabit/second (Gbps) switched ethernet, 100 Megabit/ second (Mbps) switched ethernet, and 10 Mbps non- switched ethernet. Figure A-8 shows the hosts in the testbed and how they are interconnected. The hosts are interconnected with 10/100/1000 Mbps Ethernet switches from Extreme Networks. It is important to note that multicast routing is not being used in the testbed. In this environment, all multicast and broadcast traffic is forwarded across all collision domains.

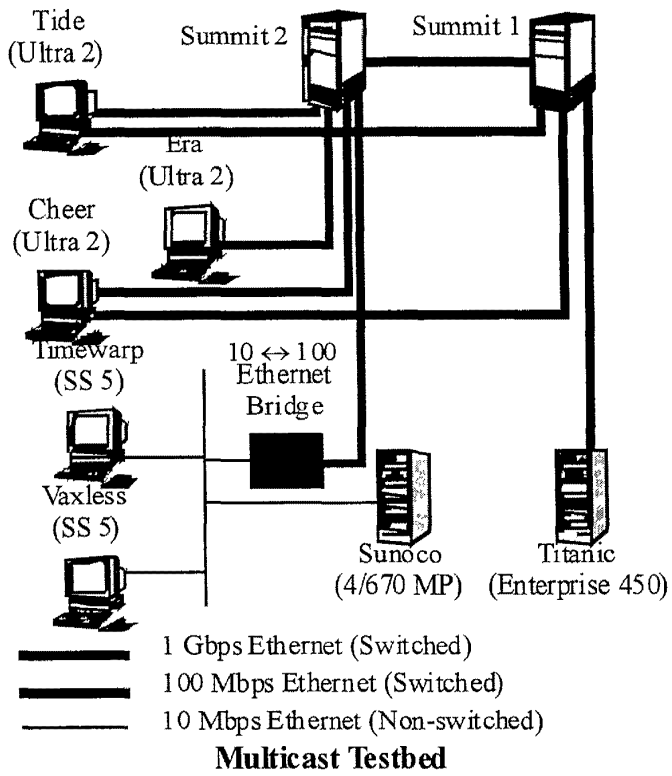


Figure A-8 Multicast Testbed

A.6.1.2 Stand-alone Unicast Versus Multicast

a. As described in Section A.4, local metrics are applicable to measuring both unicast and multicast performance. MCAST is able to report these metrics for both cases since it examines IP addresses passed to it to determine if they are multicast addresses or unicast addresses. If an address is a unicast address, the system calls associated with enabling multicast functionality are not made. In this case, the system calls made are identical to those made for unicast data exchange. If a multicast address is passed to MCAST, it makes the socket and system calls which enable the multicast functionality.

b. In this series of tests, Tide is the transmitter and Cheer is the receiver. First a unicast data stream is measured between the hosts. Next, a multicast data stream is measured between the hosts. The unicast and multicast results are then compared. For these tests, only *LET* and *LOWL* are examined.

A.6.1.2.1 100 Mbps Switched Ethernet

As can be seen in Figures A-9 and A-10, there is little if any difference between Unicast and Multicast transmission using 100 Mbps switched Ethernet with Tide as the transmitter and Cheer as the receiver.

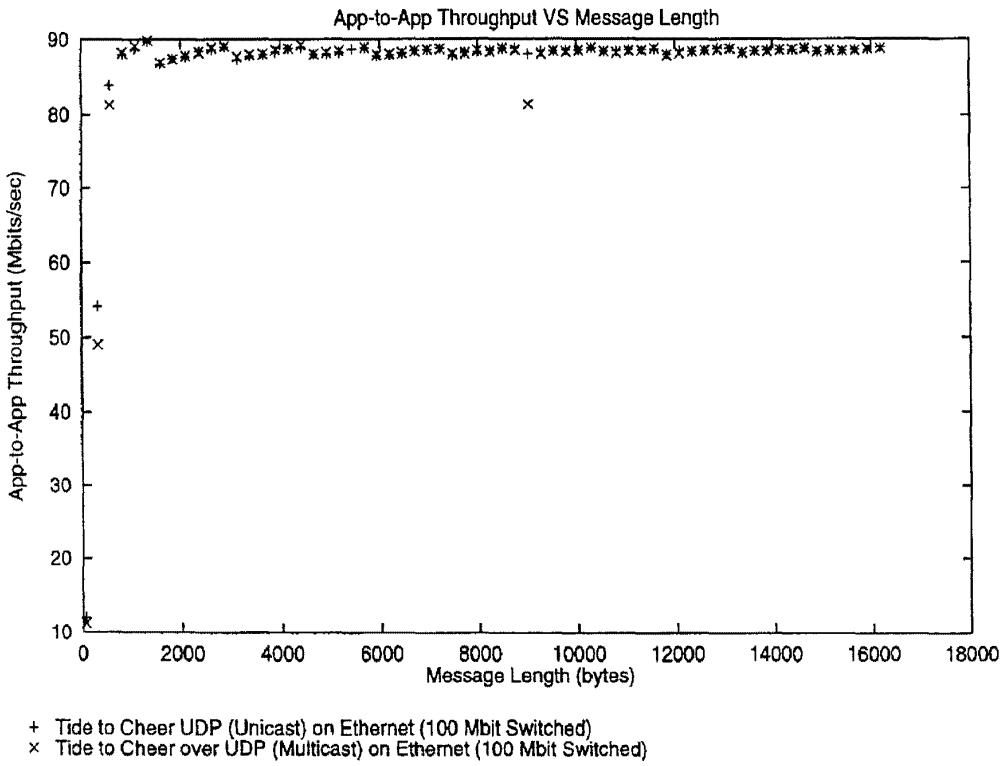


Figure A-9 Unicast Versus Multicast LAT over 100 Mbit Switched Ethernet

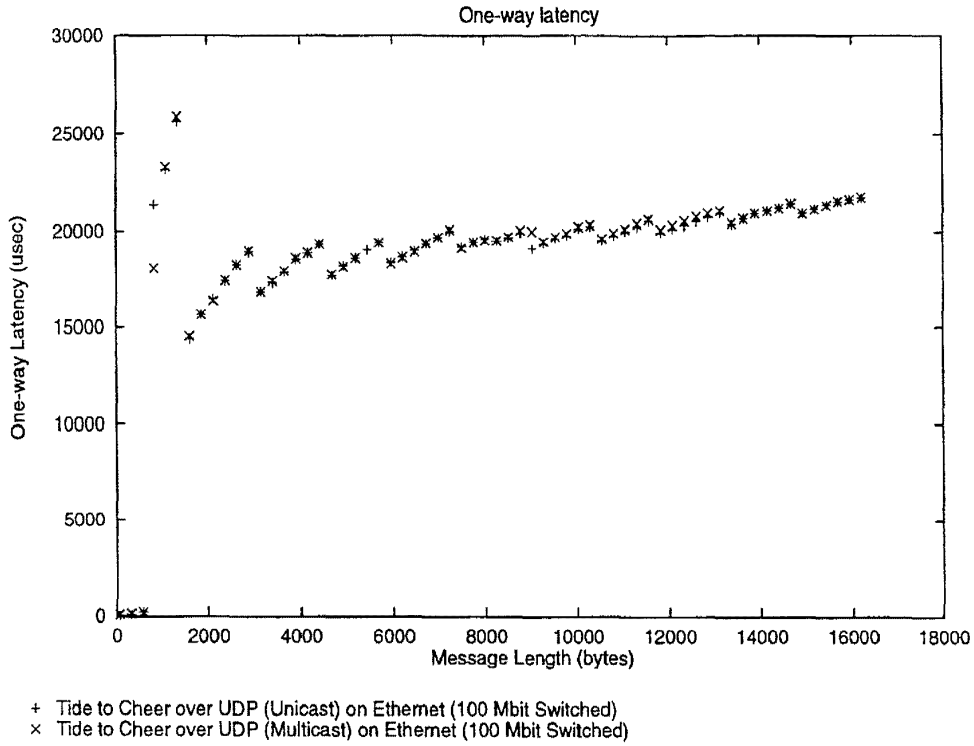


Figure A-10 Unicast Versus Multicast LOWL over 100 Mbit Switched Ethernet

A.6.1.2.2 1 Gigabit/Second (Gbps) Switched Ethernet

a. As shown in Figures A-11 and A-12, there is a noticeable performance difference in the *LAT* and *LOWL* metrics measured with Tide as the transmitter and Cheer as the receiver over Gigabit Ethernet. Figures A-13 and A-14 plot the difference between the two *LAT* and *LOWL* performance curves to better illustrate the difference between the unicast and multicast performance.

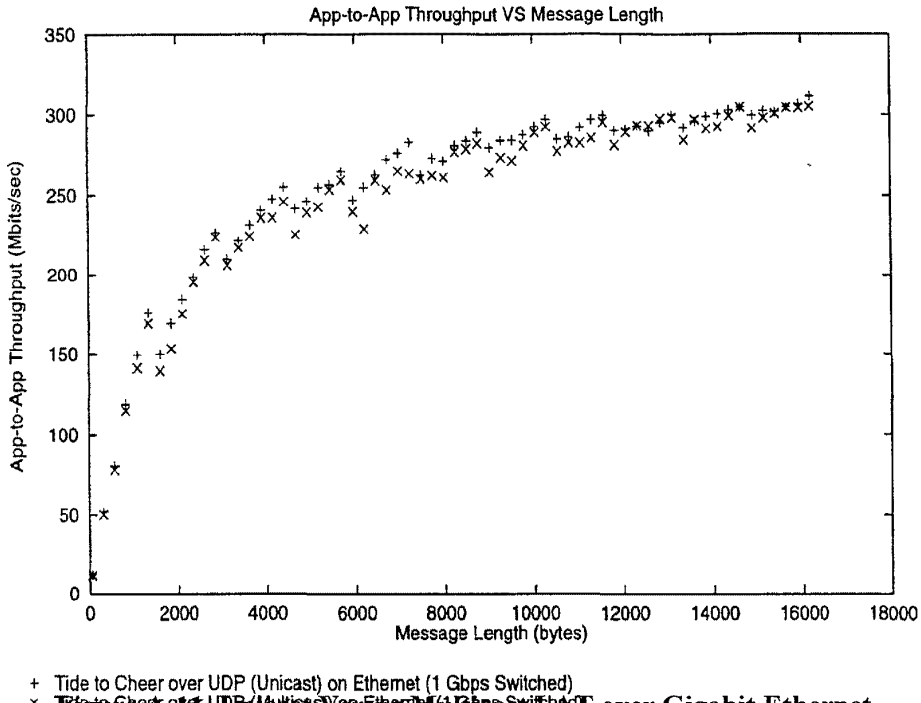


Figure A-11 Unicast Versus Multicast LAT over Gigabit Ethernet

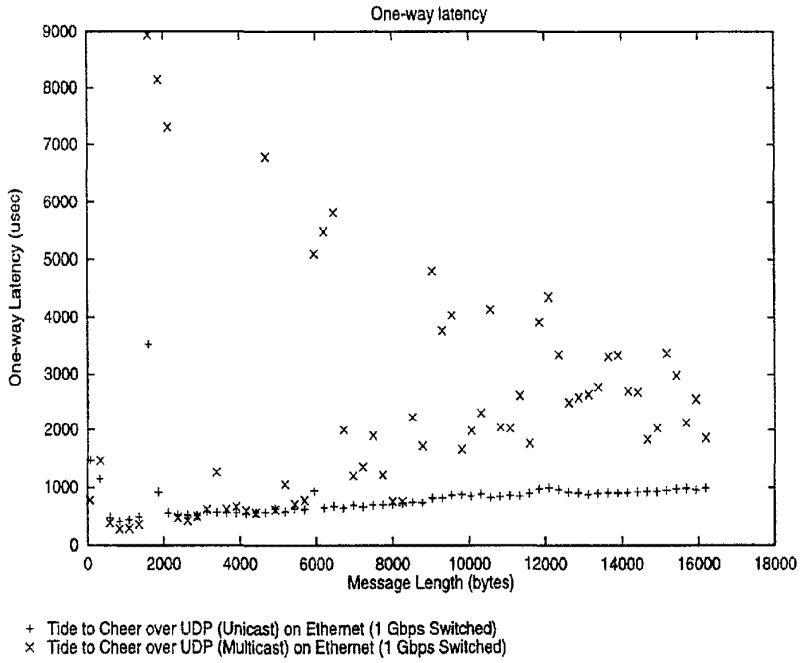


Figure A-12 Unicast Versus Multicast LOWL over Gigabit Ethernet

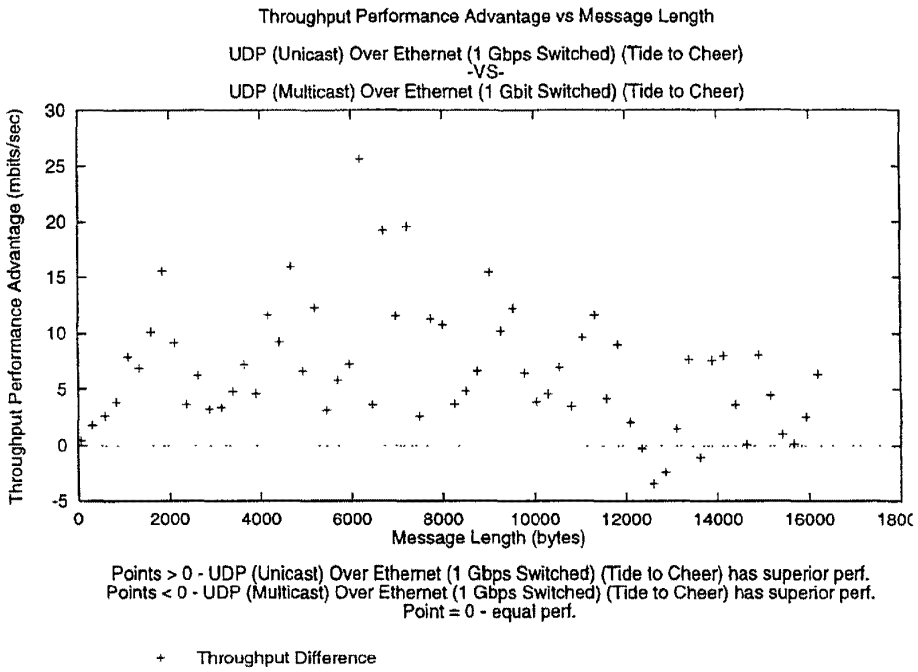


Figure A-13 Unicast and Multicast LAT difference over Gigabit Ethernet

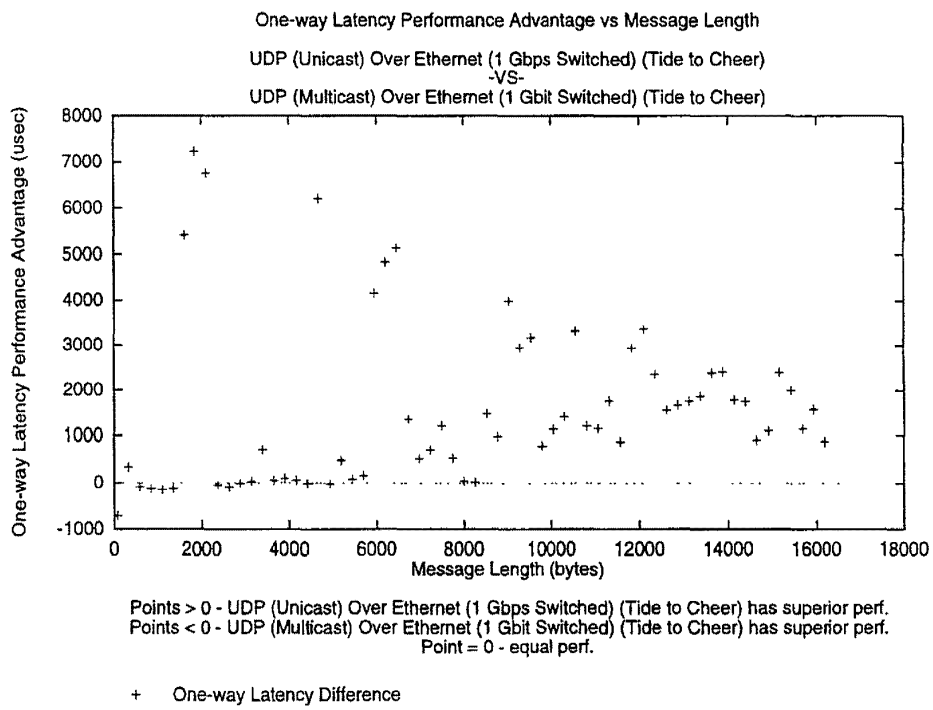


Figure A-14 Unicast and Multicast LOWL difference over Gigabit Ethernet

b. At this time, no satisfactory explanation for the performance differences observed between Unicast and Multicast transmission in the 1 Gbps tests has been found. The fact that no differences were observed in the 100 Mbps tests but differences were observed in the 1 Gbps tests might lead to the hypotheses that the multicast forwarding capabilities of the switch do not scale in the same way as the unicast forwarding of the switch at higher transmission rates. Another hypothesis would be that architectural differences between the Summit 1 switch used in the 1 Gbps tests and the Summit 2 switch used in the 100 Mbps contribute to the performance differences observed. Further work in this area will attempt to test these hypotheses and attempt to identify the source of the performance differences.

A.6.1.3 Stand-alone Group Multicast Performance

As previously noted the local metrics are useful to measure both unicast and multicast performance. The group performance metrics, on the other hand, are generally only used to measure multicast performance. When analyzing the behavior of a multicast group, the *GRC* metric is often a key in the analysis process.

A.6.1.3.1 Using GRC for Group Performance Analysis

a. Figures A-15, A-16, and A-17 show three different experiments, E1, E2, and E3 respectively, in which Sunoco was the multicast transmitter and Era, Tide, Timewarp, and Vaxless were the multicast receivers. In the tests, Sunoco transmitted 10,000 messages to the multicast group. As shown in Figure A-8, the transmitter in this scenario (Sunoco) is connected to the network via a 10 Mbps shared Ethernet connection. Consequently, Sunoco cannot exceed a transmission rate of 10 Mbps.

b. Figure A-15 shows a scenario (E1) where Sunoco sent messages of length 16384 bytes to the multicast group with a sleep_time equal to 0. As can be seen visually, few messages are not received by any of the receivers. Table A-1 shows a high degree of correlation (.9991) among the receiver set and that a high percentage of the messages were received (99.87%).

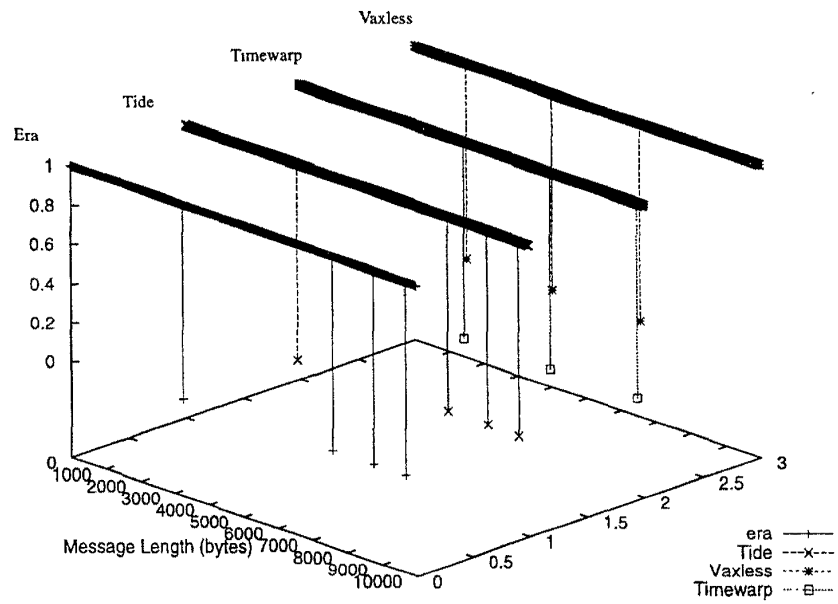


Figure A-15 E1: Reception Vectors for Message 16384 Bytes in length (no background load)

Table A-1 GRC Measurements

	GRC	RR=0	RR=N	0<RR<N
E1	0.9991	0%	99.87%	.13%
E2	0.9989	1.81%	97.99%	0.2%
E3	0.4498	0%	21.06%	78.94%

c. Figure A-16 shows a scenario (E2) where Sunoco sent messages of length 16384 bytes to the multicast group with a sleep_time equal to 0. In this scenario, unlike E1 and E3, a unicast background load of 6 Mbps (between two hosts not involved on the multicast test) was introduced on the 10 Mbps Ethernet shown in Figure A-8. As can be seen, in Figure A-16, a large number of messages are dropped. Table A-1 shows a high degree of correlation among the receiver set (0.9989). It also shows that nearly all the messages which were not received were not received by all receivers (RR=0). In this environment, a message not received by all hosts probably indicates that the message was lost at the sender. With the background load on the shared media attached to the sender, it is likely that collisions caused it's output queue to overflow resulting in lost packets.

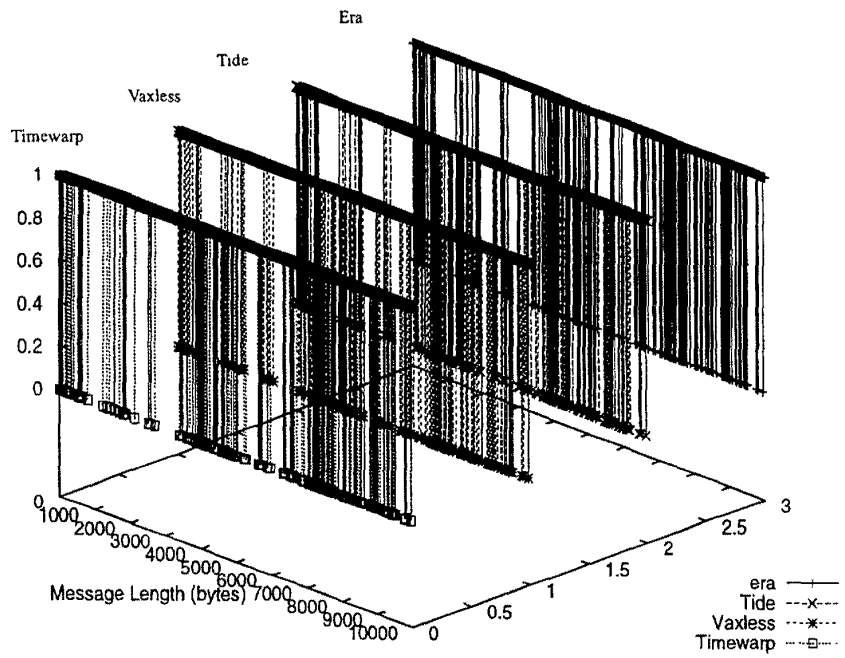


Figure A-16 E2: Reception Vectors or Messages 16384 bytes in length (6 Mbps background load)

A.6.1.3.2 GRC Performance Partitioning

Figure A-17 shows a scenario (E3) where Sunoco sent messages of length 256 bytes to the multicast group with a sleep_time equal to 0. For E2, Table A-1 shows a much lower degree of correlation (.4498) than E1. This is also reflected in the Reception Reports (RR) in the table. Computing Π for E3 with $\tau = 0.5$ yields the set $\{\{R_1,R_2\}, \{R_3, R_4\}\}$ where R_1, R_2, R_3, R_4 are the reception reports for Tide, Era, Vaxless, and Timewarp respectively. Consequently, Tide and Era correlate to the specified degree as do Vaxless and Timewarp.

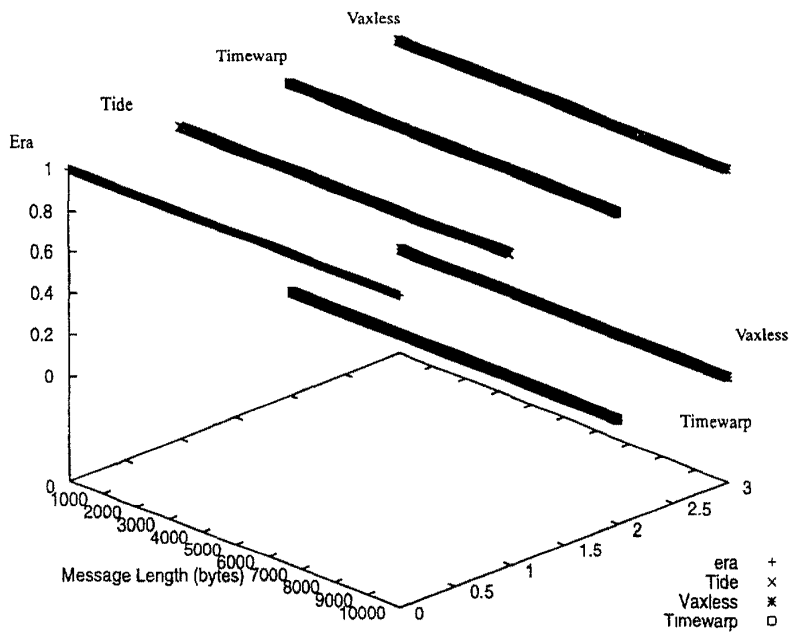


Figure A-17 E3: Reception Vectors for Messages 256 bytes in length (no load)

A.6.2 Integrated Multicast Testing

MTS proved useful in diagnosing several unexpected performance behaviors in the HiPer- D testbed. These behaviors were operational in nature and thus of significant importance. These problems needed to be solved as quickly as possible to ensure successful operation of the testbed.

A.6.2.1 ATM Switch "Performance Problem"

a. When an ATM Switch "Performance Problem" was being investigated, the complete MTS was not available. The only multicast instrumentation tool available was MCAST (which would later become part of MTS). MCAST was used to try to duplicate the performance problem being experienced by the applications in the testbed using its application performance simulation tuning parameters.

b. Duplicating the problem with the test tool was important since this would show that the problem was not necessarily in the applications experiencing the performance problems. Also, producing the problem required a significant portion of the HiPer-D testbed to be activated along with the applications which ran on those portions of the testbed. This process is very resource intensive in both the equipment and personnel required. If the problem could be duplicated with MCAST, a single operator could reproduce the problem at will by using only the machines on which the applications experiencing the performance problems were run.

c. The initial attempts to duplicate the problem proved unsuccessful because of a lack of understanding of the actual performance characteristics of the applications being simulated by MCAST. Once the performance characteristics of the applications were understood, duplicating the problem proved to be quite easy and was repeatable.

d. When the performance problem was duplicated a great deal of analysis was needed to determine the actual cause. Since the MTS test coordination tool, data extraction tool, and data analysis tools were not developed yet, running tests was both tedious and time consuming. Also, analyzing the data was difficult as well. Despite these problems, the source of the problem was found without the tools.

e. A lesson learned from this process was that better testing, instrumentation, and analysis tools were needed. This realization led to the full development of MTS and the formalized development of the metrics presented in this appendix.

f. When the MTS implementation was complete, another attempt was made to duplicate the "performance problem". The goal of this experiment was to determine if the additional tools developed would reduce the complexity in debugging the problem. Unfortunately, all of the conditions required to reproduce the problem could not be duplicated. The utility of the new toolset was still shown, however, because a new "performance problem" was identified.

A.6.2.2 Altair4 Performance Problem

In trying to duplicate the ATM switch "performance problem", a test was conducted which included all of the Altair hosts (Sun Microsystems Ultra2 workstations). This test revealed a new performance problem. Plots of the data extracted by using mcast_extract.pl showed the *LAT* and *LPMR* performance of Altair4 was consistently less than that of the other Altair hosts. Furthermore the *GRC* computed was poor. The *GRC* data was analyzed with the grc_partition.pl tool which placed Altair4 in a partition by itself and all other hosts in a second partition. Since all of the Altair hosts have identical hardware and software configurations, this result was not expected.

A.6.2.3 Altair and Electra Performance Differences

In another experiment conducted to duplicate the switch "performance problem", all of the Altair hosts (Sun workstations) and Electra hosts (SGI Origin 200 workstations) were included in the test. The data collected showed that all of the Altair hosts had correlated performance while the Electra hosts did not. In the testbed, all the Electra hosts are connected to one switch and all the Altair hosts are connected to another switch. Since the LANE Broadcast Unknown Server (BUS) is located on the switch connecting the Altair hosts, it was speculated that the intra-switch traffic required for the Electra hosts to participate in the group was contributing to the problem. Another experiment was run in which several Electra hosts were moved to the switch with the LANE BUS. Since these hosts still exhibited the performance problem, it was concluded that the problem was not switch related. It is speculated that performance differences between the Electra and Altair hosts would account for the differences in the correlation. Further testing is needed in this area.

A.7 CONCLUSIONS

The metrics and analysis techniques developed in this appendix as well as their realization in MTS have proved to be useful in analyzing the performance of multicast systems. The performance of complex applications can be simulated and evaluated on the target hardware environment without the complexity of running the application itself.

A.8 FUTURE WORK

a. The development of MTS will be completed. The data extraction tool will be updated to include all local and group metrics presented herein.

b. The work to date has focused on instrumenting IP Multicast in a non-routed environment. Future work in this area will examine a routed multicast environment and a switched VLAN environment. Further investigation into the source of the performance differences between 1 Gbps and 100 Mbps ethernet will be conducted.

c. Another area of future work would be to investigate incorporating the metrics defined here into an operational tool which can be used to monitor the health of an operational network. The MCAST receiver module would be converted to a daemon process and run on hosts to be monitored.

A.9 REFERENCES

[1] [BMWG] Dubray, K., Internet-Draft: Terminology for IP Multicast Benchmarking, July 1997.
[2] [IREY97] Irey IV, Philip M., Marlow, David T., Harrison, Robert D., Distributing Time Sensitive Data in a COTS Shared Media Environment, Joint Workshop on Parallel and Distributed Real-Time Systems, pp. 53-62, April 1997.†
[3] [IREY98] Irey IV, Philip M., Harrison, Robert D., Marlow, David T., Techniques for LAN Performance Analysis in a Real-Time Environment, Real-Time Systems - International Journal of Time Critical Computing Systems, Volume 14, Number 1, pp. 21-44, January 1998.†
[4] [IPPM] Paxon, V., Almes, G., Mahdavi, J., Mathis, M., Internet Draft: Framework for IP Performance Metrics, November 1997.
[5] [NTP] Mills, D., RFC-1305, Network Time Protocol (Version 3) Specification, Implementation and Analysis, March 1992.
[6] [RFC1112] Deering, S., RFC-1112, Host Extensions for IP Multicasting, August 1989.

†Documents available on http://www.nswc.navy.mil/ITT

APPENDIX B

HIPER – D
DATA DISTRIBUTION EXPERIMENT

Leslie A. Madden, Robert B. Anthony, Charles L. Fudge
Naval Surface Warfare Center, Dahlgren Division
Dahlgren, Virginia 22448-5000
{MaddenLA, AnthonyRB, FudgeCL}@nswc.navy.mil

Ex.1009 / Page 194 of 280
TESLA, INC.

APPENDIX B

HIPER – D DATA DISTRIBUTION EXPERIMENT

Leslie A. Madden, Robert B. Anthony, Charles L. Fudge
Naval Surface Warfare Center, Dahlgren Division
Dahlgren, Virginia 22448-5000
{MaddenLA, AnthonyRB, FudgeCL}@nswc.navy.mil

B.1 BACKGROUND - DATA DISTRIBUTION PROBLEM DOMAIN

a. The data distribution problem domain for command and control systems can be divided into two general categories: control data and streaming data, as shown in Figure B-1. Complex data ordering, low volume, reliable delivery, and deterministic latency often characterize control data. Examples of this type of data for the Aegis Combat System could be Auto-Special and Doctrine data.

DATA DISTRIBUTION			
STREAMING		**CONTROL**	
Best Effort	Reliable/Acknowledge	Reliable Acknowledge	
Volume High or Low		Volume and Rate Low	
Rate High or Low		Ordered	Deterministic

Figure B-1 Data Categories

b. Streaming data usually has limited data order dependencies, high volume, requires a stable frequency and inter-arrival, and does not require reliable delivery. A message can be missed, or possibly several messages depending on the frequency and message type, the next received, and still satisfy system requirements. Examples of this type of data could be Gyro and Track file update data.

c. Obviously, not all the message types used in the Aegis Combat system fall neatly into one of these two categories. SPY data, for example, could be considered high volume and yet require a stringent deterministic latency.

d. The relevance of these complex issues is very germane to future Aegis baselines. Baseline 7 Phase 1 timing requirements are currently being reviewed on a message type basis. Tradeoffs for example, such as deterministic latency versus reliable delivery, are being evaluated to determine the most efficient method of distributing data to meet the system's requirements.

e. Future Aegis baselines will use distributed processing architectures and distributed applications that employ commercial off the shelf hardware and software to the greatest extent possible. Software companies have developed, and continue to develop, a broad variety of commercial middleware products to help system developers implement distributed applications

in the data distribution arena. One group of products that appears to meet the data streaming requirements is publish/subscribe products.

f. This paper documents the results of the evaluation of two publish/subscribe products.

B.2 PUBLISH/SUBSCRIBE PRODUCTS

a. Publish/Subscribe products are message-based middleware products that provide a service, or range of services, to various applications, above and beyond those provided in TCP/IP. Applications that need to publish data send it to the publish/subscribe product. The product in turn sends the data to any and all clients that have registered with it, for the data. As shown in Figure B-2, a publisher can service multiple subscribers, and a network could consist of multiple publishers and subscribers. The benefit of the publish/subscribe product is that the developer doesn't have to design and develop the code, and can concentrate on the development of applications. Second, the application doesn't have to be involved in keeping track of clients and the sending and possible re-sending of data. Third, the developer can use the same or similar interfaces in all of their servers and clients that utilize the same types of publish/subscribe service. Basically it is less costly to buy a messaging middleware product than it is to build one.

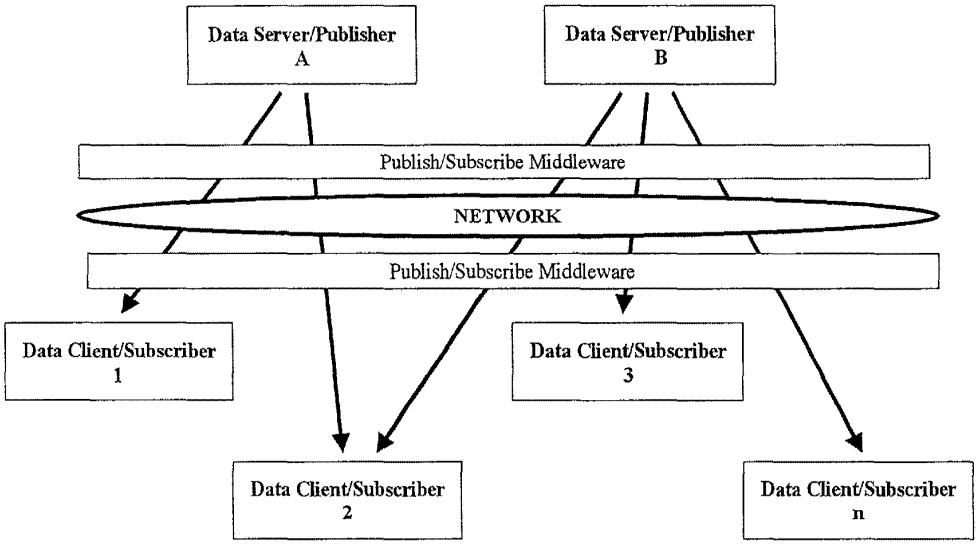


Figure B-2 Publish/Subscribe Architecture

b. In its simplest form, a publish/subscribe product requires the following functions:

(1) **Subscriber Registration** – Subscribers/clients need to be able to register and un-register with the publisher for services from the publisher. Typically this is accomplished with a daemon that runs in the background.

(2) **Publisher Advertisement** – Publishers need to be able to advertise the presence of their services. Typically this is accomplished with a daemon that runs in the background.

(3) **Data Publishing** – When the application actually publishes its data, the publish/subscribe product must take the data and distribute it via the network to all clients that have registered for the data.

(4) **Data Reception** – The data is received by the publish/subscribe product in each of the clients, which in turn provide it to the subscriber application.

c. While these are the basic functions provided in a publish/subscribe product, there are an endless number of additional features that can be added, such as reliability of delivery, fault tolerance, different rates of delivery, language bindings, etc.

B.3 GOALS AND OBJECTIVES

a. The primary goals of the Data Distribution Experiment (DDE) were to:

(1) evaluate two Publish/Subscribe middleware products in a data streaming application.

(2) select one product to be used for Navigation Data Distribution for the Advanced Computing Testbed Demonstration 98 (Demo 98)

b. The two products selected for evaluation were Real Time Innovation's NDDS and Tibco's Rendezvous (TIB).

c. The objectives of the DDE were to:

(1) modify the GDCSim and GDC_Client programs to incorporate the NDDS and TIB products

(2) modify the GDCSim and GDC_Client architectures to accommodate future middleware products

(3) install the middleware products in the test bed, and

(4) compare NDDS and TIB performance; and compare both to UDP performance in the following areas:

(a) Average Message Latency

(b) Message Latency Variation

(c) Average Message Inter-arrival time

(d) Message Inter-arrival Variation

(e) Dropped message rate

(f) ATM network versus FDDI network

(g) Unicast versus Multicast

(h) Memory utilization

(i) CPU utilization

B.4 EXPERIMENT DESCRIPTION

B.4.1 Experiment Design

a. The DDE architecture is shown in Figure B-3. The basic idea was to have a publisher application on one workstation transmit messages to a subscriber application on another workstation. Successive test runs were made utilizing UDP, followed by inserting NDDS as the middleware, and finally with TIB inserted as the middleware. Test data was gathered for the parameters listed above, and then used to compare the three products. UDP was used as a baseline for several reasons. First, it is a fast protocol in widespread usage. Second, it is the protocol that both NDDS and TIB use as their underlying transport protocol. As a result, we expected both NDDS and TIB performance to be "worse" than UDP.

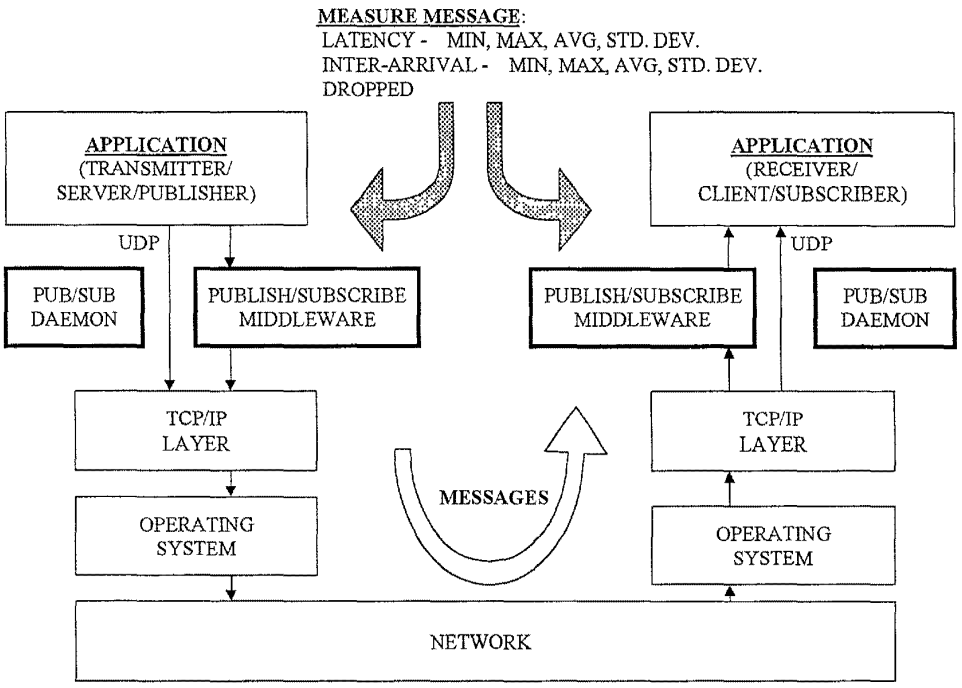


Figure B-3 Data Distribution Experiment Architecture

b. Ideally, once a message stream between the publisher and the subscriber has been established, one would expect the only variable to be the middleware. Unfortunately, this is not true. Each of the components shown in Figure B-3 inherently inserts some variation in the message stream. Therefore, it was very important to minimize the variation each component added, and/or keep the variation the same for successive runs.

B.4.1.1 Definition Of Latency And Inter-Arrival Variation

a. Most of the parameters being measured, such as latency, dropped message rate, etc, are easily understood. The purpose of this section is to describe how we are using the latency variation and inter-arrival time variations to discriminate between UDP, NDDS and TIB.

b. Using inter-arrival time as an example, Figure B-4 shows the ideal frequency distribution for a publisher. All of the messages would have an inter-transmit time of exactly the desired message rate, in this case, 1.0 msec.

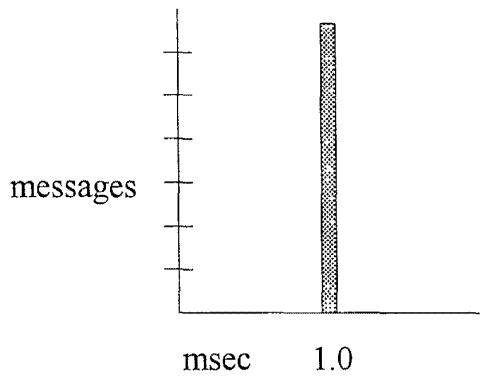


Figure B-4 Ideal GDCSim Frequency Distribution

c. Assuming that we are using UDP, the hypothetical data in Figure B-5 shows that, at the subscriber, we would expect most messages to arrive with 1.0 msec inter-arrival times. However, we would also expect a certain number of messages to arrive either a little late or a little early because they are affected by the TCP/UDP/IP layers, operating systems and network layer of the test set-up. This is reflected in Figure B-5, as 0.9 and 1.1 msec. inter-arrivals.

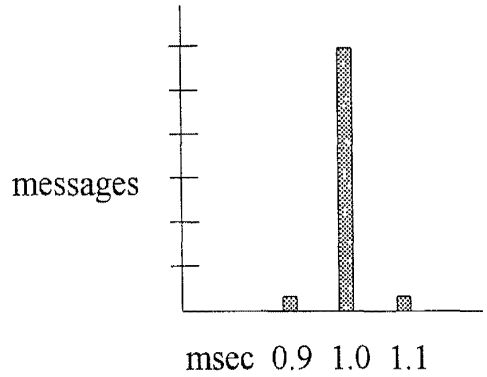


Figure B-5 GDC_Client UDP Frequency Distribution

d. This leads us to several very important concepts.

(1) UDP represents the baseline for any measurements and/or comparisons between products.

(2) The combined variation of UDP, the operating systems, network, etc. cannot be removed from the experiment.

(3) Since the products we are measuring are based on UDP and TCP, their variations will almost certainly be equal to, or greater than, UDP.

e. Assuming the above issues, and assuming hypothetical test data from two different products, product "A" and product "B", Figures B-6 and B-7 show two different frequency distributions that might be seen. All things being equal, then product "A" has better inter-arrival times than product "B", and represents a better choice.

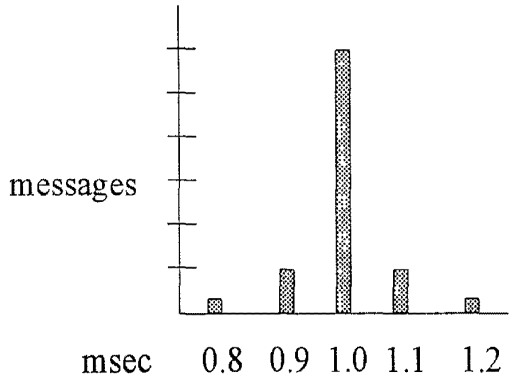


Figure B-6 GDC_Client Product "A" Frequency Distribution

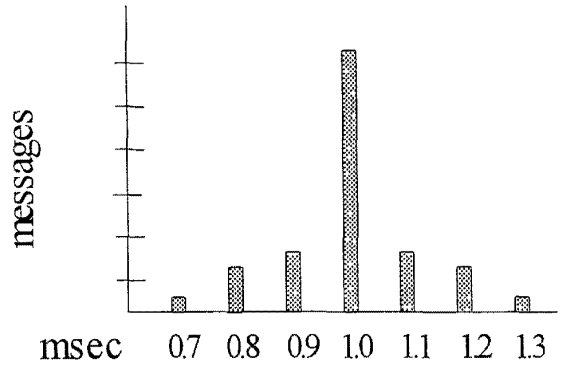


Figure -B7 GDC_Client Product "B"
Frequency Distribution

f. Standard deviation is the chosen measure of the spread, or variation, of the inter-arrival data in comparing the three publish/subscribe products. The standard deviation was chosen, first for convenience. GDCSim and GDC_Client provide a standard deviation for each 2-second sample. Second, it is a well accepted measure of variation. And third, it is relatively easy to calculate. It is very important to recognize that standard deviation is being used as a measure of variation introduced by the middleware products as compared to the variation associated with UDP. The absolute variance added by the middleware product is not being calculated. A similar approach is used for the latency variation, however, instead of the variation in inter-arrival, variation in latency is being analyzed.

B.4.2 Hardware Configuration

The DDE was conducted on three workstations and two networks as shown in Figure B-8. One set of test runs was conducted between Aquilla and Blofeld (Sun Sparc Ultra 2s) over the ATM network. Another set of test runs was conducted between Aquilla and Pavo (Sun Sparc Ultra 1s) over the FDDI network. All the test runs consisted of UDP, NDDS and TIB messages.

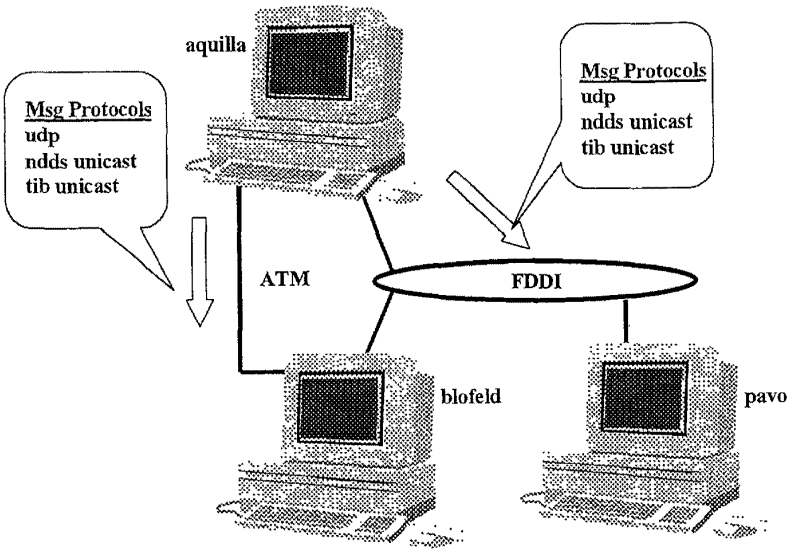


Figure B-8 DDE Hardware Configuration

B.4.3 Software Architecture

a. The high-level software architecture is shown in Figure B-9. GDCSim runs on Aquilla and the clients, GDC_Client, each run on Blofeld and Pavo. The purpose of GDCSim is to transmit constant streams of messages for various time periods. Multiple combinations of message rate and time periods can be set up in a configuration file that GDCSim reads in. For example, 100 messages per second may be transmitted for 2 minutes, followed by 500 messages for 2 minutes, etc. In addition to sending messages, GDCSim registers with the Isis server protos, and transmits control messages to the clients, after they register with GDCSim via protos. One of the major modifications to GDCSim consisted of making a "general purpose" interface layer where various types of messages could be added for testing. The current message types supported at this layer are UDP, TCP, Isis, NDDS and TIB. The idea was to make it easy to add additional message types as needed. Note that there are also "network services" layered into the system, specifically Isis, NDDS and TIB, which in turn make use of UDP, TCP and IP as necessary. Another key point is that there are NDDS and TIB daemons on the workstation that are required for the messaging products to work. The purpose of the daemons is to "coordinate" the registration of services between the middleware layer in the GDC_Clients, and the middleware layer in the GDCSim.

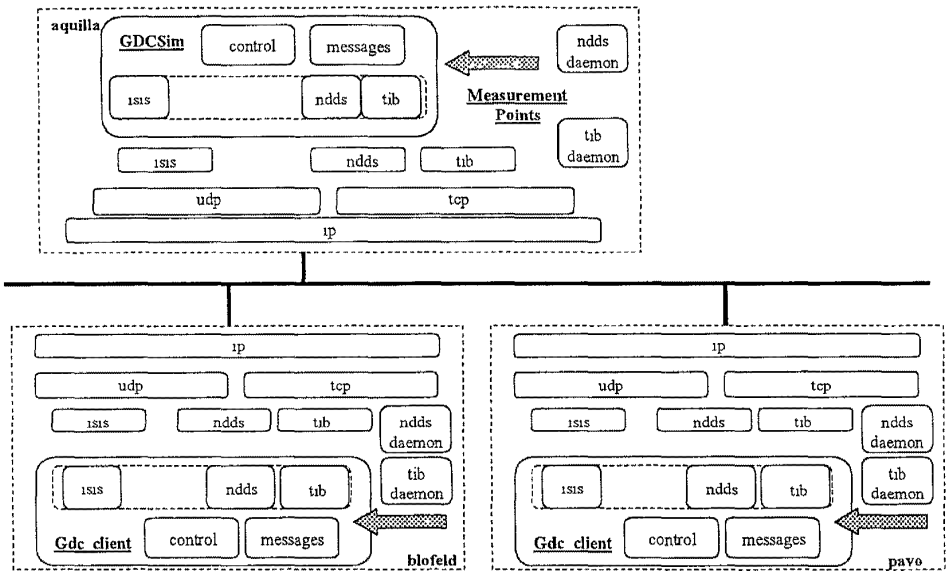


Figure B-9 Software Architecture

b. When GDC_Sim generates a data message, it is immediately passed to the middleware layer. The middleware layer then passes the message to lower network protocol layers and out onto the network. At the receiving workstation, the process is reversed, the lower network protocol layers pass the message to the middleware layer, which then passes the message to GDC_Client.

B.4.4 Test Scenario And Conditions

a. The data transmission protocols were selected and read in via initialization files and command line arguments. The UDP protocol was used to gather baseline data for data transmission. The messaging middleware products, NDDS and TIB/Rendezvous, were also used for data transmission in a unicast mode, and then compared to UDP. The underlying protocol that NDDS uses for data and daemon communication is UDP. TIB/Rendezvous uses TCP daemon communication and UDP for data transmission. Each product was tested during two runs, an A run and a B run. Each test run was identical and consisted of the various message rates as shown in Figure B-10. Latencies, interarrival times, standard deviations, etc. were collected in two-second intervals, as shown in Figure B-11.

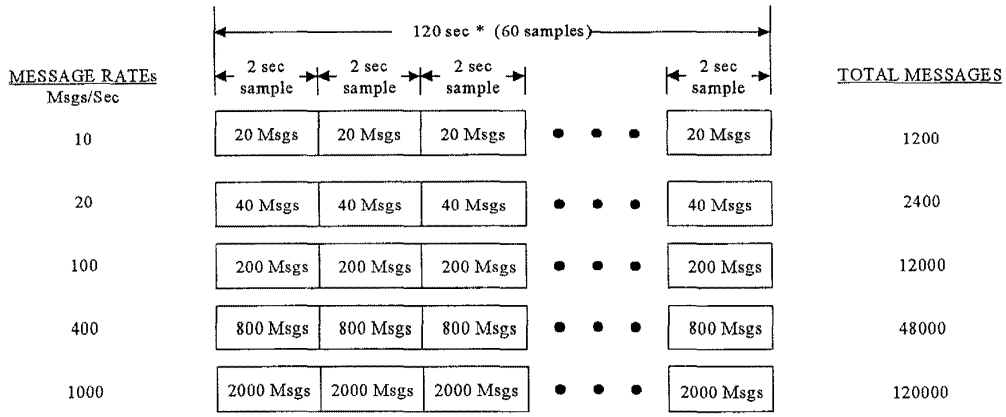


Figure B-10 Message Rates

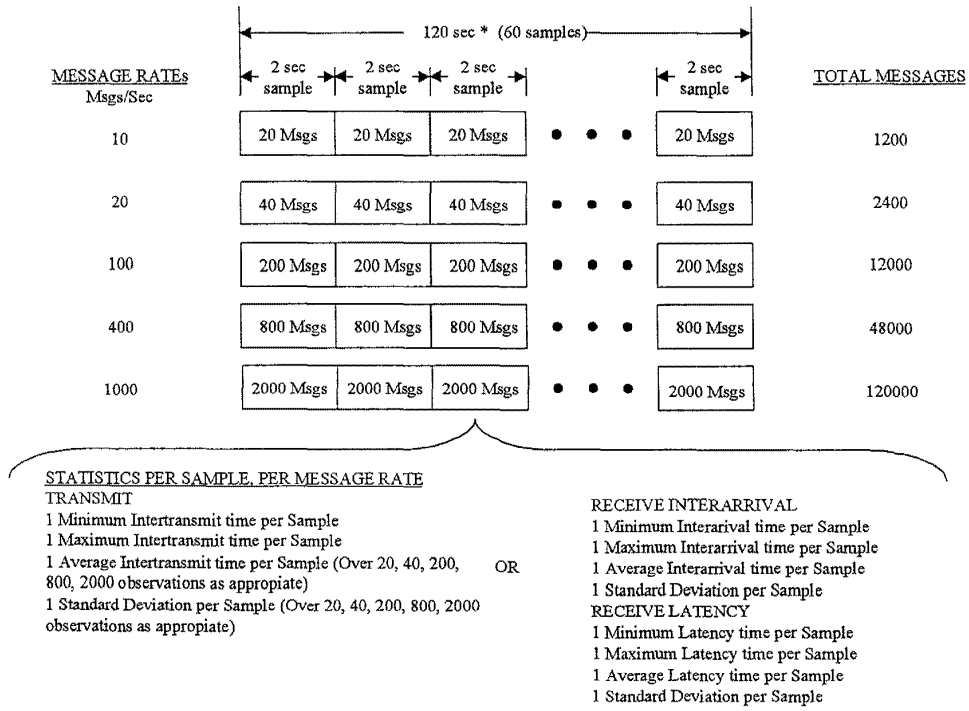


Figure B-11 Statistics Per Sample

b. The message transfer rates "requested" by GDC_Client were also read in via an initialization file. The message length was 200 bytes of canned data. It should be noted that gyro data for Aegis Baseline 7 Phase 1 is being proposed at a 100-Hertz rate.

c. Sun Sparc Ultra 1 (Pavo) and Ultra 2 (Aquilla, Blofeld) workstations were used with the Solaris 2.6 operating system. Sun Sparc Ultra 1 is a uniprocessor workstation and the Ultra 2 contains two processors. ATM and FDDI network interfaces were used for data transmission.

B.5 ERROR MITIGATION AND ANALYSIS

a. The purpose of the error mitigation and analysis effort was to take a rigorous approach to identifying, quantifying, and reducing the errors and uncertainties encountered in the data collection and subsequent analysis. The end goal of this effort was to minimize the effect of errors and uncertainties on the final results and conclusions.

b. Multiple test runs were conducted to "smooth out" and identify any perturbations with data associated with an individual test run. The System Control Laboratory (SCL) was physically isolated, and only those applications necessary to conduct the testing were running. The Solaris OS real time feature was used to place GDCSim, GDC_Client, and the NDDS and TIB/Rendezvous daemons in real-time mode. Both this and the isolation of the SCL were done to minimize the uncertainties associated with the latency and inter-arrival data.

c. During the runs, when GDCSim was crossing from one frequency to another, data that was obviously invalid was identified and characterized by latencies or inter-arrivals inconsistent with the frequency, but consistent with the previous frequency. Such data was not considered in the final analysis.

d. The Network Time Protocol (NTP) was used for clock synchronization. Blofeld and Pavo, the two clients were synchronized to Aquilla (GDCSim) which was synchronized to Cetus. Table B-1 lists the clock data collected at the beginning and end of each test run. All values are in microseconds.

Table B-1 Clock Synchronization Values

	AQUILLA		BLOFELD		PAVO	
	Run A	**Run B**	**Run A**	**Run B**	**Run A**	**Run B**
NDDS - Start	002	238	000	008	-088	-023
NDDS- Finish	-010	075	145	024	179	087
UDP - Start	039	-098	-035	100	011	041
UDP - Finish	528	-108	072	-019	043	-028
TIB - Start	-060	048	-047	057	-006	027
TIB - Finish	210	655	004	112	-026	117

e. "Start" and "Finish" data was obtained manually before beginning each test run, and at the end each test run. From an absolute value perspective it can be seen that the greatest clock

drift occurred between – 088 and 179 for NDDS Run A on Pavo. It drifted at least 267 microseconds. At its worst, there was a 179 usecs deviation from Aquilla. However, it should be noted that, as it drifted through zero, it was virtually synchronized with Aquilla. The rest of the runs fell under 160 usecs.

f. It is obvious that lacking absolute synchronization of clocks between the sender of the data, Aquilla, and the receivers of the data, Pavo and Blofeld, results in measurement error through any algorithms used to calculate latency, thus affecting the final value. But the magnitude of error possible has, at least, been bounded with the data in Table B-1; that it is relatively small and, most importantly, does not change the conclusions reached in the experiment.

g. For a number of reasons there was no attempt to assign a numerical value to this error of uncertainty, or to add any positive or negative offsets to the final latency measurements. First, the clock drift rates, the NTP resynchronization rate, and the message sample rate are all occurring at different rates. Instrumentation and methodology do not currently exist to dynamically measure and factor in clock offsets. However, as long as NTP maintains clock synchronization within acceptable levels it is probably not necessary.

B.6 TEST RESULTS

B.6.1 Latency

GDC_Client collected the latency data for messages received in a two-second window. The average latencies for each message frequency rate were then averaged via a spreadsheet. The end-to-end latency measurement was the difference between the GDCsim time stamp and the GDC_Client time stamp. Figure B-12 illustrates how the latency and messages are related.

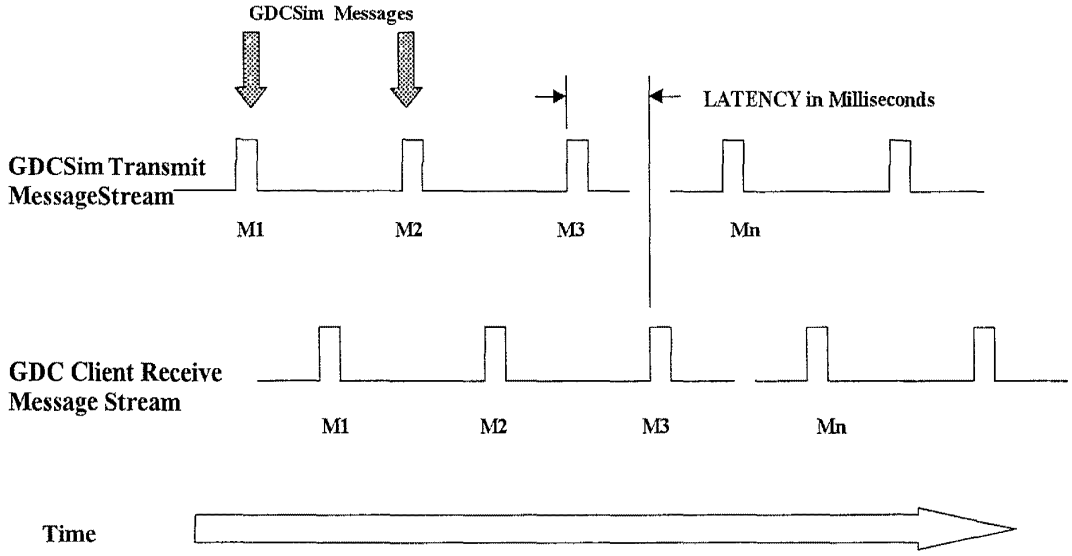


Figure B-12 Message Latency

B.6.1.1 Average Latency

Figures B-13 and B-14 show the average latency for Blofeld (ATM) and Pavo (FDDI) respectively. The UDP and NDDS latencies are relatively constant across the frequency spectrum on both the ATM and FDDI networks. The TIB/Rendezvous latency is relatively constant across all message rates for ATM however, it increases 25 to 30% at the 1000 message per second rate for FDDI. As the baseline protocol, UDP has the best performance, as would be expected. For the messaging middleware products, NDDS has a better absolute latency performance than TIB/Rendezvous. It should be noted that all average latencies fell within the 3.5 msec. range which is the latency specified for Mission Critical Messages in the Preliminary Design Review Data Package for Computing System Requirements Document for Baseline 7 Phase 1.

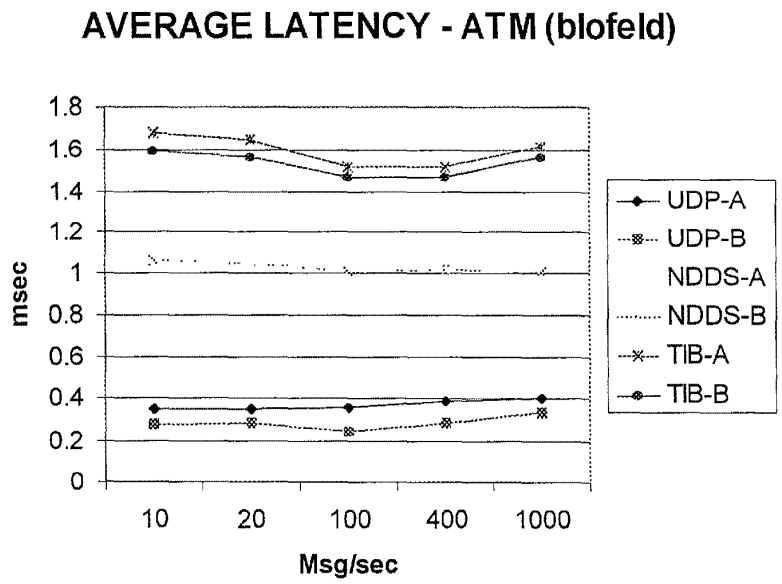


Figure B-13 Average Latency Results, Blofeld

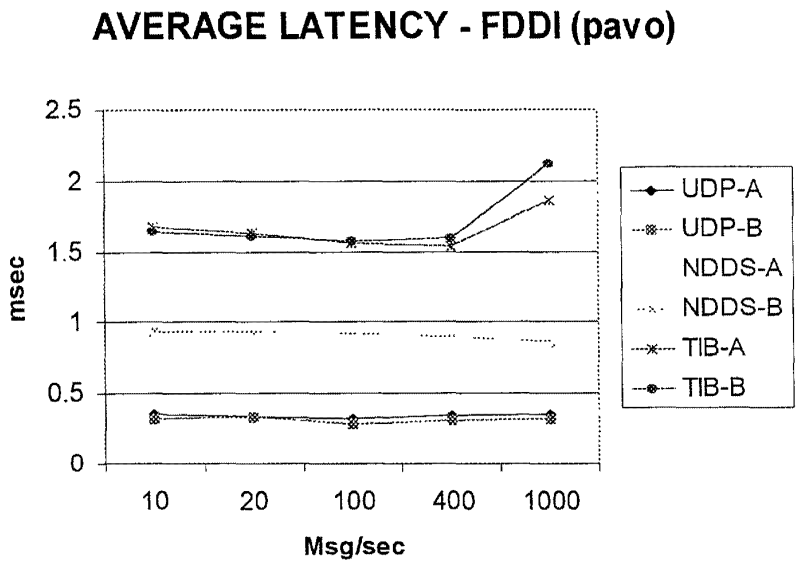


Figure B-14 Average Latency Results, Pavo

B.6.1.2 Latency Variation

Figure B-15 and B-16 show variation in latencies, as measured by the standard deviations, for each frequency. The standard deviation captures the variation in latency measurements with respect to the mean. The average standard deviation for UDP and NDDS are consistent throughout the frequencies and relatively small. The standard deviation for TIB/Rendezvous begins to climb dramatically at 400 messages per second. This indicates the product is beginning to have a significant number of messages with relatively high latencies.

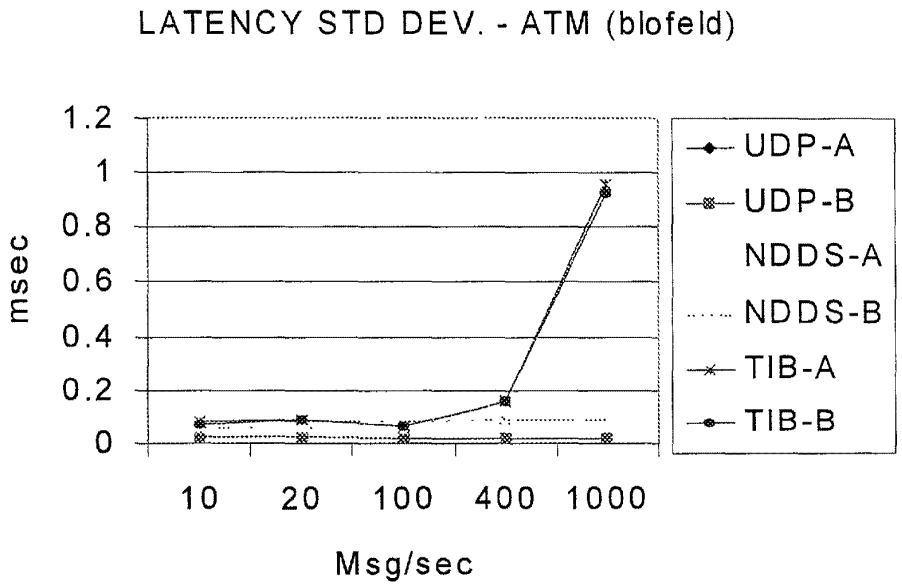


Figure B-15 Latency Variations, Blofeld

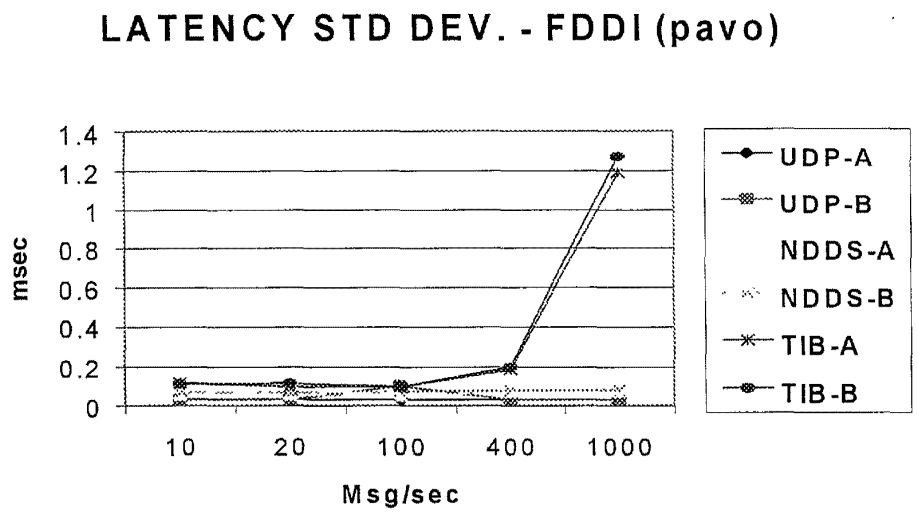
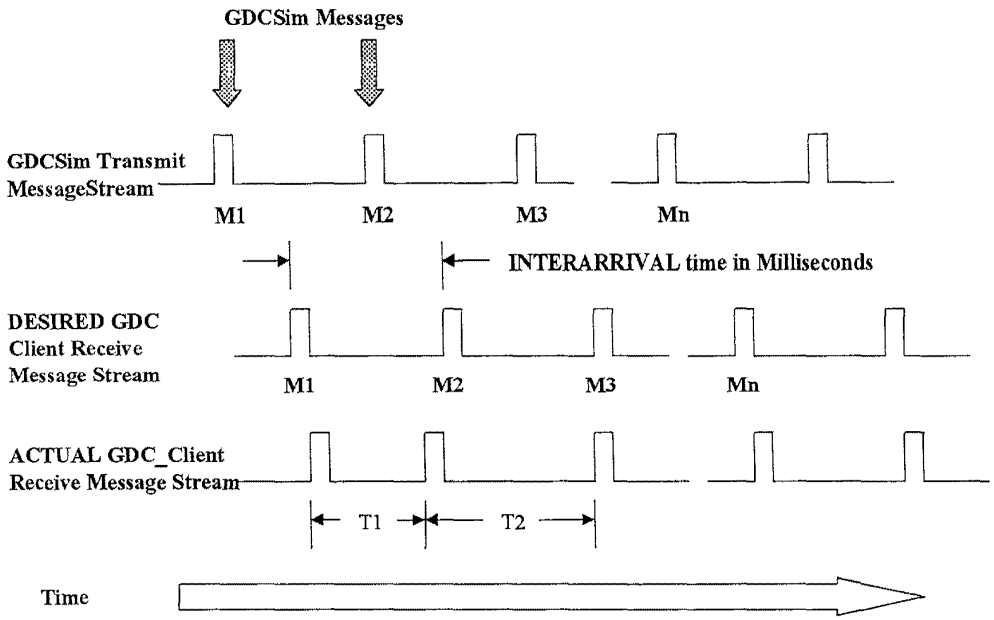


Figure B-16 Latency Variations, Pavo

B.6.2 Inter-Arrival

One characteristic of a publish/subscribe middleware product is its ability to deliver messages at a constant rate or period. In the DDE, GDC_Client has the ability to measure the time between each successive message, which is called the inter-arrival time. Figure B-17 shows the relationship between an idealized GDCSim message stream, what we would like to see at the GDC_Client, and what the message stream may actually look like. As discussed earlier, the time measurements are made when the message leaves the lower level network layers and actually enters the GDC_Client application.



Constant Interarrival Times - Reflect the ability of the "system" to adhere to a given message rate, message frequency, and conversely, a constant period.

Figure 17 Inter-arrival Time

B.6.2.1 Average Inter-Arrival

a. GDC_Client collected the inter-arrival data for messages received in each two-second window. The average inter-arrival times for each message frequency rate were then averaged via a spreadsheet. Figure B-18 shows the relationship between message rate and average inter-arrival times over the ATM network, and Figure B-19 shows the relationship between message rate and average inter-arrival times over the FDDI network.

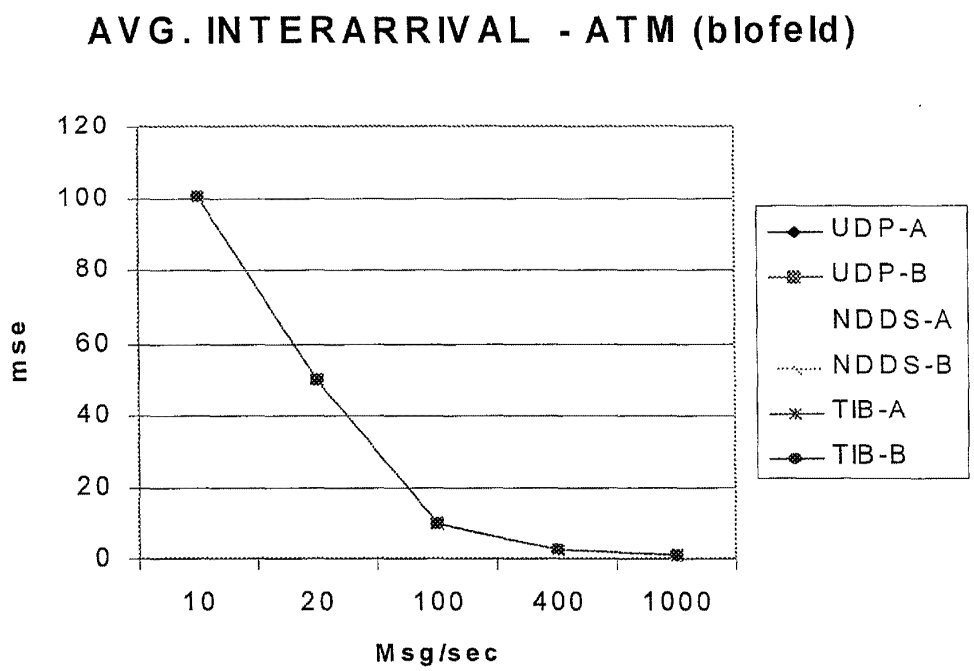


Figure B-18 Average Inter-arrival Time - Blofeld

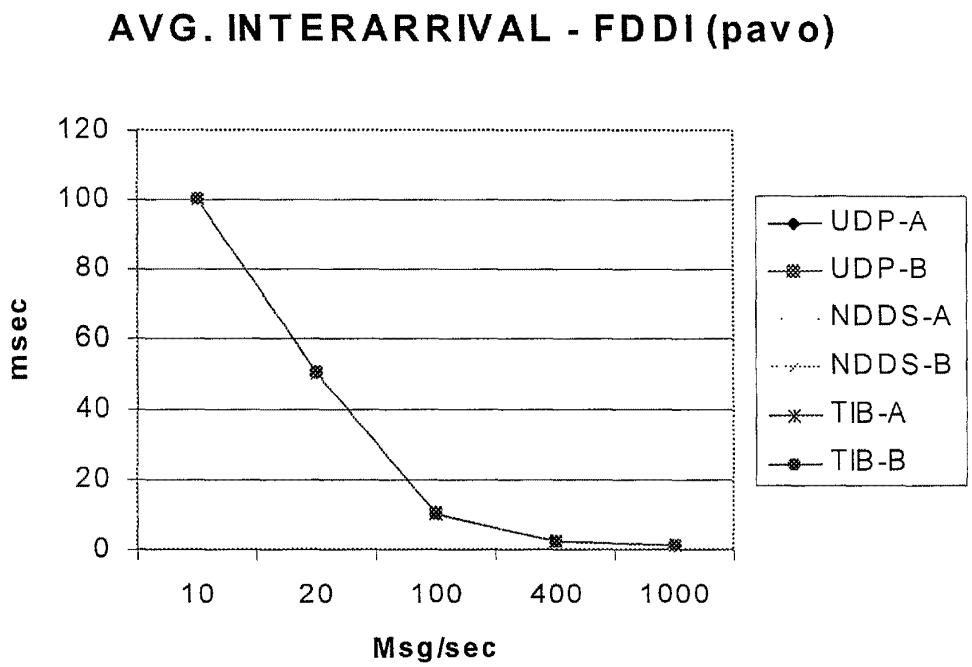


Figure B-19 Average Inter-arrival Time - Pavo

b. As shown by the overlapping plots all of the protocols, UDP, NDDS and TIB, were capable of producing and receiving message streams that achieved the desired average message rates. Notice that the results were virtually identical for both the ATM and FDDI interfaces.

c. As result, if the only interest is average inter-arrival times, all three of the products considered in the DDE are acceptable.

B.6.2.2 GDCSIM Transmit Variations

a. Before looking at inter-arrival variation times at the GDC_Clients, it is necessary to look at how well GDCSim could actually transmit messages. Ideally, GDCSim should transmit messages at exactly the same inter-transmit time; in other words, at a very constant rate. Unfortunately, GDCSim itself has a small but significant variation in inter-transmit times. This variation will contribute to the total variation that is seen and measured at the GDC_Client. As a result, some effort must be made to quantify its variation and analyze its impact on the experiment.

b. Figure B-20 shows the variations in inter-transmit times at various message rates, over the ATM network. Figure B-21 shows the inter-arrival time at the GDC_Client. In general, at each respective message rate the standard deviations of the inter-arrival times are at least twice the size of the inter-transmit times.

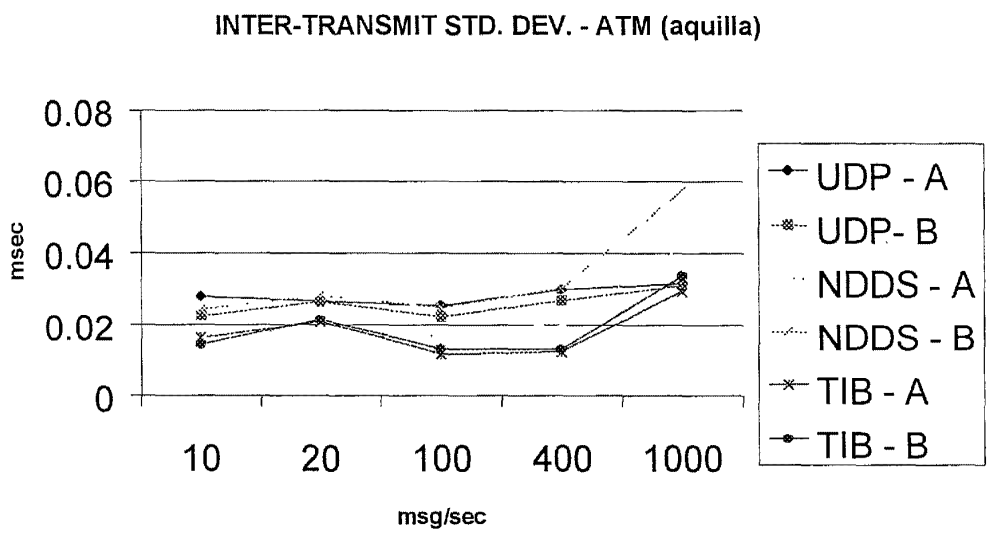


Figure B-20 Inter-Transmit Standard Deviation Aquilla

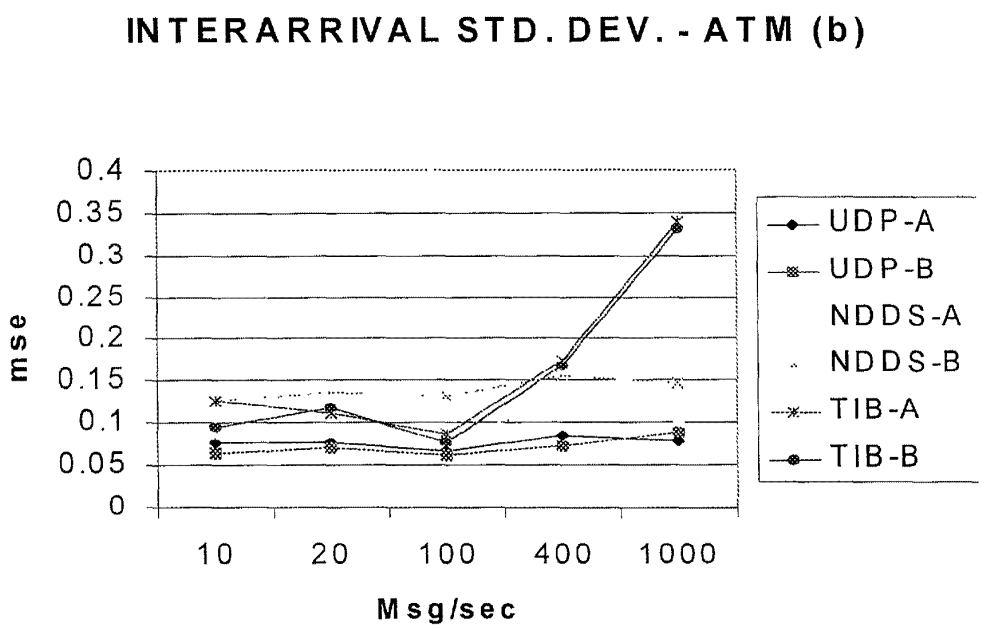


Figure B-21 Inter-arrival Variation – Blofeld

a. Figure B-22 shows the variations in inter-transmit times over the FDDI network, and Figure B-23 shows the inter-arrival time at the GDC_Client. Once again, at each respective message rate, the standard deviations of the inter-arrival times are at least twice the size of the inter-transmit times.

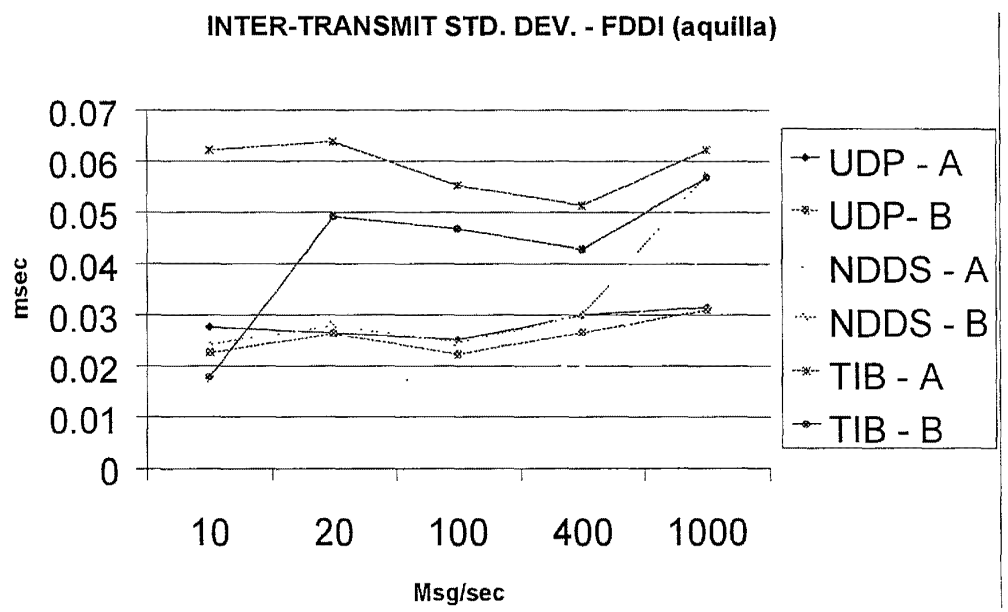


Figure B-22 Inter-Transmit Standard Deviation - Aquilla

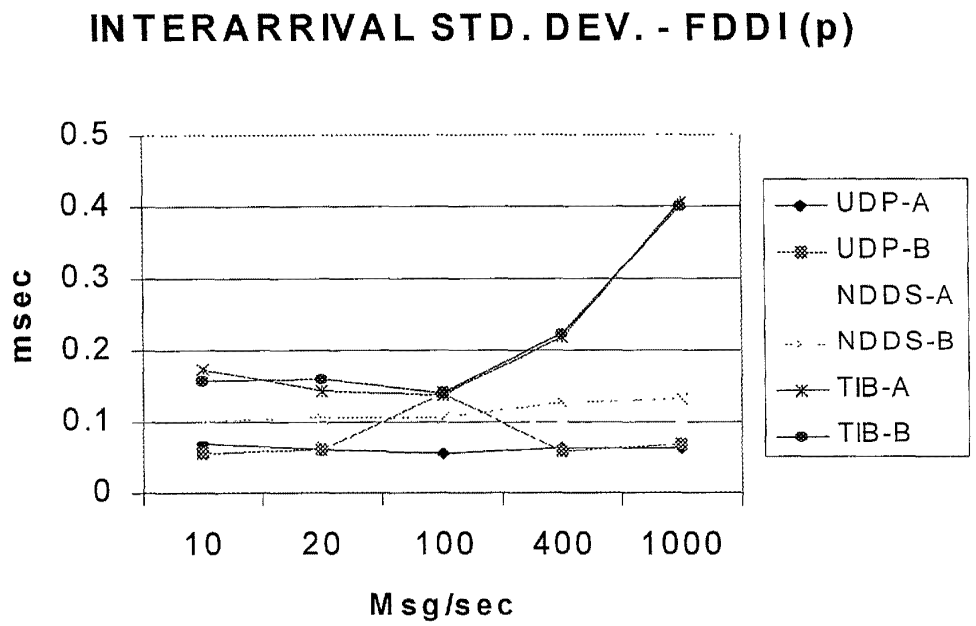


Figure B-23 Inter-arrival Variation – Pavo

b. Looking at the UDP protocol, while the inter-transmit variation is significant, there is also a significant component of variation in the inter-arrival variation that can be attributed to the TCP/IP layers, operating systems and the network associated with the testbed. This variation is inherent to the testbed and cannot be removed. The UDP plots are relatively flat and well behaved over all the message rates.

c. For NDDS, the data shows pretty much the same results as UDP, with the exception that there is a small rise in inter-transmit variation at the 1000 msg/sec rate. Interestingly enough, there was no corresponding rise observed in the inter-arrival variation.

d. Finally, for TIB, the inter-transmit times are small but not very consistent between the different networks. At the clients, the inter-arrivals are fairly flat, until we get to the 400 and 1000 msg/sec rates, where the variations get significantly larger. The magnitude and rate of change at these rates are very significant.

e. In both the case of NDDS and TIB, there appears to be a significant component of inter-arrival variation above and beyond what can be attributed to GDCSim, or the other components such as TCP/IP, the operating systems, network, etc. Since everything in the test system is held reasonably constant, this variation is attributed to the middleware products.

B.6.2.3 Inter-Arrival Variation

a. Figure B-20 shows the inter-arrival variation for each message rate at Blofeld. The standard deviation captures the variation in inter-arrival measurements with respect to the mean. The average standard deviation for UDP and NDDS are consistent throughout the frequencies and relatively small. The standard deviation for TIB/Rendezvous begins to climb significantly at 400 and 1000 messages per second. This indicates the product is beginning to have a significant number of messages arriving either late or early. Figure B-23 shows identical results at Pavo over the ATM network.

b. There are two interesting yet conflicting things happening with TIB. While the standard deviation is rising rapidly, it is indicating that a proportionately small number of messages are having significant deviations. As shown earlier, the average inter-arrivals are very good. On the other hand, the direction of the growth is ominous.

B.6.3 Dropped Message Rate

Dropped message rate data was gathered to identify any differences in performance between UDP, NDDS and TIB. Over all the test runs, there were no dropped messages.

B.6.4 ATM Network versus FDDI Network

As expected, both the ATM and FDDI networks produced the same results. The only real numerical differences were in average latency, which were minor differences in TIB at 1000 msg/sec over the FDDI network.

B.6.5 Multicast

Multicast testing was not performed because Pavo did not have an ATM interface, and because the multicast configuration over the FDDI was not working at the time of the test runs.

B.6.6 Memory Utilization

a. Table B-2 illustrates the typical memory values recorded during the testing. The application programs, GDCSim and GDC_Client were consistently around 7 to 8 MB for UDP, NDDS and TIB/Rendezvous. There were two NDDS daemons and a license manager running on one workstation, plus a start daemon which ran on all workstations. The NDDS daemons were consistently around 2 MB.

b. The TIB/Rendezvous daemon on the clients was approximately 2 MB also. The Rendezvous daemon on the server grew from 2 MB to 37 MB in approximately 120 seconds from the start of the test. A default feature of TIB/Rendezvous is that it provides 60 seconds of reliability. It accomplishes this by buffering the last 60 seconds worth of data. According to discussions with the vendor, "37 MB is not out of the ordinary, unless you are observing

persistent growth over long period of time". Persistent growth over a period of time was not observed.

Table B-2 Memory Allocation

APPLICATION/DAEMON	START	END
GDCSim (Aquilla)	8 MB	7 MB
GDC_Client (Blofeld)	8 MB	7 MB
GDC_Client (Pavo)	8 MB	8 MB
NDDS Daemons (Server and Clients)	2 MB	2 MB
TIB Rendezvous Daemon	2 MB	2 MB
TIB Rendezvous Daemon (Server)	2 MB	37 MB

B.6.7 CPU Utilization

CPU utilization data was gathered, but was not analyzed due to time constraints.

B.7 COMBINING STANDARD DEVIATIONS ACROSS SAMPLES

a. As discussed earlier, GDCSim and GDC_Client calculated the standard deviation for both latency and inter-arrival time for each 2-second sample of data. (Note, that GDCSim and GDC_Client, collected data on every message sent. As a result, data is available on the entire population of any given test run.) To compare UDP, NDDS and TIB at the various message rates, the variation across all the samples at a given message rate was needed as shown in Figure B-24. This section details the mathematical approach taken to combining the separate variations into a single variation at each message rate.

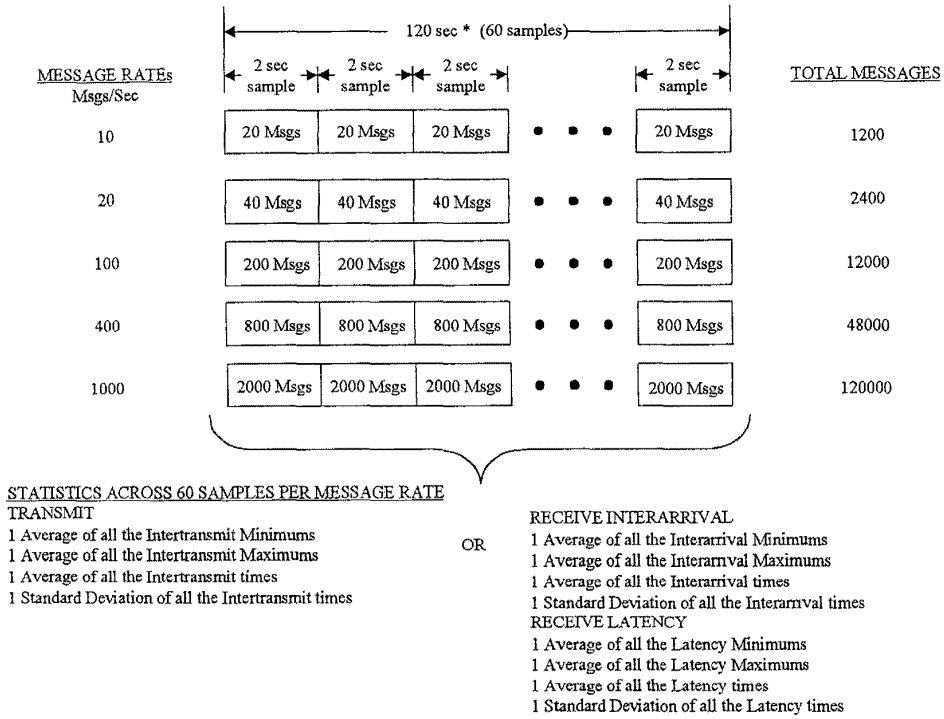


Figure B-24 Statistics Across All Samples

b. The calculation for the standard deviation across all the samples at a given message rate is possible, because the averages of the individual samples are all extremely close. The standard deviation is calculated as follows.

Individual Sample Variance Calculations

The formula for calculating the variance is as follows:

$$s^2 = \frac{n\left(\sum_{i=1}^{n} x_i^2\right) - \left(\sum_{i=1}^{n} x_i\right)^2}{n(n-1)} \qquad \text{(Equation 1)}$$

Where each individual measurement is x_i

For large values of n, the variance can be defined as:

$$s^2 = \frac{n(\sum_{i=1}^{n} x_i^2) - (\sum_{i=1}^{n} x_i)^2}{n(n)} \qquad \text{(Equation 2)}$$

(It is important to note, that the assumption of a large values of n is valid for 100, 400 and 1000 messages per second, because we have 200, 800 and 2000 observations respectively, per sample. However, the assumption is not true for message rates of 10 and 20 because they result in 20 and 40 observations per sample. The same formulas were used for calculating all of the standard deviations because the absolute standard deviations at these message rates represent an extremely small value when compared to the actual values of the latencies and interarrivals being measured. Secondly, the interesting deviations in latency and inter-arrival all occurred at the 400 and 1000 messages per second rates.)

$$s^2 = \frac{\sum_{i=1}^{n} x_i^2}{n} - \left(\frac{\sum_{i=1}^{n} x_i}{n} \right)^2 \qquad \text{(Equation 3)}$$

For an individual sample of n observations:

$$s_j^2 = \frac{\sum_{i=1}^{n} x_{ji}^2}{n} - \left(\frac{\sum_{i=1}^{n} x_{ji}}{n} \right)^2 \quad \text{for } j = 1, 2, \ldots m \qquad \text{(Equation 4)}$$

$$s_j^2 = \frac{\sum_{i=1}^{n} x_{ji}^2}{n} - \bar{x}_j^2 \qquad \text{(Equation 5)}$$

$$\text{a)} \qquad s_j^2 + \bar{x}_j^2 = \frac{\sum_{i=1}^{n} x_{ji}^2}{n} \qquad \text{(Equation 6)}$$

(The above equation is used in a later substitution.)

Overall sample variance

The overall sample variance is

$$S^2 = \frac{\sum_{j=1}^{m}\sum_{i=1}^{n} x_{ji}^2}{mn} - \left(\frac{\sum_{j=1}^{m}\sum_{i=1}^{n} x_{ji}}{mn}\right)^2 \qquad \text{(Equation 7)}$$

$$S^2 = \frac{\sum_{j=1}^{m}\sum_{i=1}^{n} x_{ji}^2}{mn} - \left(\frac{\sum_{j=1}^{m} \bar{x}_j}{m}\right)^2 \qquad \text{(Equation 8)}$$

Substituting in equation (a) from above,

$$S^2 = \frac{\sum_{j=1}^{m} s_j^2 + \bar{x}_j^2}{m} - \left(\frac{\sum_{j=1}^{m} \bar{x}_j}{m}\right)^2 \qquad \text{(Equation 9)}$$

if $\bar{x}_j = \bar{x}$ (all of the averages are very close to each other), then

$$\sum_{j=1}^{m} \bar{x}_j = m\bar{x} \qquad \text{(Equation 10)}$$

and

$$S^2 = \frac{\sum_{j=1}^{m} s_j^2}{m} + \frac{m\bar{x}^2}{m} - \left(\frac{m\bar{x}}{m}\right)^2 \qquad \text{(Equation 11)}$$

$$S^2 = \frac{\sum_{j=1}^{m} s_j^2}{m} \qquad \text{(Equation 12)}$$

c. The above formula was used to combine the standard deviations across multiple samples for latency, inter-arrival and inter-transmit variations.

B.8 EXPERIMENT CONCLUSIONS and SUMMARY

a. NDDS was selected to be used for Navigation Data Distribution for Demo 98 for the following reasons:

(1) NDDS had the best absolute latencies

(2) NDDS had smaller variations in latency and inter-arrival times

(3) TIB appears to be reaching its limits at 1000 msgs/sec based on growing variations in latency and inter-arrival times observed

b. All three products, UDP, NDDS and TIB demonstrated latencies within the 3.5 msec definition in the Preliminary Design Review Data Package For Computing System Requirements Document For Baseline 7 Phase 1.

c. Two commercial messaging products were integrated into the GDCSim/GDC_Client and evaluated. The programs were modified to accommodate other products in the future. The products are viable products for use based on their proper application.

d. The publish/subscribe products performed essentially the same over both ATM and FDDI networks.

e. As expected, both NDDS and TIB functioned well in the testing. Even though NDDS was chosen over TIB, TIB is a perfectly acceptable product for use within the limits of its capability.

B.9 LESSONS LEARNED

a. The following lessons were learned:

(1) Installing publish/subscribe middleware products requires a significant amount of effort. There are so many features in these products that close attention has to be paid to installation to ensure that any evaluation is fair to both products.

(2) It was possible to discriminate between products based on average latencies and variations in latency and inter-arrival times.

(3) Efforts to evaluate products at rates above 1000 msgs/sec will have to pay special attention to clock synchronization and the magnitude of variations inherent in the testbed.

(4) The shape of the latency and inter-arrival frequency distributions should be characterized.

B.10 FUTURE EFFORTS

a. The current experiment was a "raw speed" test. Future evaluations should be more like the "real world" environment and utilize more of the basic features of the products.

b. Also, testing with multiple clients would give a better idea of the scalability of the product.

c. Perform multicast testing

APPENDIX C

WINDOWS NT INVESTIGATIONS

The Johns Hopkins University
Applied Physics Laboratory
Laurel, Maryland 20723-6099

APPENDIX C

WINDOWS NT INVESTIGATIONS

The Johns Hopkins University
Applied Physics Laboratory
Laurel, Maryland 20723-6099

C.1 Background

In June 1998, the Navy's Chief Information Officer (CIO) released his Information Technology Standards Guidelines (ITSG). The guidelines recognized the growing presence of Windows NT, as well as its possible application to the Navy's requirements for a powerful operating system. The ITSG indicated an organization-wide shift towards NT over the next few years, encompassing systems such as ashore and on-ship installations. The question now was not, "Should we use NT?" but "What do we need to do to make NT work the way we want?" The focus of the activities outlined here was to begin investigations that would ultimately answer the latter question for HiPer-D.

C.2 Objectives and Overview

a. Our primary objective was to create an advanced distributed system, running on NT that could be used to evaluate the suitability of NT to these types of systems. JHU/APL's HiPer-D components constitute a system of this type. All of JHU/APL's HiPer-D components are written on top of a JHU/APL-developed middleware layer called Amalthea. Amalthea isolates the application from the specifics of the underlying operating system. Moving the system to NT was accomplished by porting Amalthea to NT.

b. Amalthea also contains a communications layer that isolates the applications from the underlying ISIS process group communications system that formed the backbone of HiPer-D. Stratus, the vendor of ISIS, is removing ISIS from the commercial market. This prompted a separate activity to move Amalthea communications away from ISIS and on to the AL. This would isolate Amalthea communications from the underlying group communications product. This effort is described in Appendix E. The decision was made to port the AL to NT rather than investing effort in the short-lived ISIS environment. The process group communications package chosen to replace ISIS was Spread. Spread runs under Unix and NT and was used to allow the components of HiPer-D that ran on NT to interoperate with the components that remained on Unix. The HiPer-D components ported included KINED, SIMCON, SENSIM, and the gen_*[1] applications.

c. Once a working hybrid system was in place, JHU/APL's next objective was to optimize the performance as much as possible. NT is generally believed to have a less efficient networking subsystem as compared to Unix, so a drop in performance was expected. However,

[1] The gen_* components are used to enter and delete tracks manually in HiPer-D.

JHU/APL had to determine exactly how much of a loss would be incurred and whether performance would remain at acceptable levels.

C.3 Steps Taken

Migration of the HiPer-D system to NT involved two major tasks: porting the underlying Amalthea API and configuring the Spread group communications package to run on NT. These two tasks were conducted in parallel. Because the HiPer-D applications were built using Amalthea, the task of moving those applications to NT was subsumed by the task of porting of Amalthea.

C.3.1 Deep Porting of Amalthea to Win32

a. The primary step in migrating HiPer-D to the NT platform was to port the underlying Amalthea libraries upon which most of the HiPer-D applications were built to the Win32 API. The purpose of Amalthea was to hide the underlying implementation and platform details from applications, providing uniform interfaces to thread models, event models, and the like. Amalthea was designed to allow applications using Amalthea libraries to compile and run on different platforms with *no code change*.

b. Two options were considered. One possibility was to perform a "shallow" port of the libraries using a third-party interface such as Cygnus' GNU Win32 environment. Such an interface translates Unix system calls into corresponding Win32 calls. However, this development path was not followed due to the lack of thread support in the translation interfaces.

c. It was decided instead to use the other option: perform a "deep" port of Amalthea, that is, manually translate the Posix-based calls inside Amalthea into a Win32 implementation. An effort was made to maintain a compatible Unix code base by using compiler #ifdef directives to conditionally generate code for different platforms. This directly supports the future goal of reintegrating Unix and NT versions into a single code base.

d. All code was compiled using Microsoft Visual Studio 97 C++ Compiler (SP3) and linked with multithreaded C libraries. Compiler warnings were set to their highest level. Consequently, as is common when compiling C code with C++ compilers, loose type specifications in the Unix code surfaced. All relevant Unix code was modified to use explicit type casting. This was a change in syntax to the Unix code base, not an algorithmic modification. In general, existing code was not touched in order to maintain a close match between the original Unix and new Unix/NT code bases.

e. There were enough parallels between Unix and Win32 that the majority of the porting was straightforward. However, libthea presented some interesting issues, namely in the implementation of threads, synchronization objects, signals, and events. The Win32 version was coded to simulate the required operations as well as possible. To date, only one situation in which the simulated behavior produced discrepancies has been found: The way that Win32 handles events causes some notifications to arrive at different times. The situation was corrected by verifying that the correct notification was received rather than assuming it was correct.

f. Table C-1 provides a list of the libraries and capsules that were ported to Win32. In addition, one new library was developed to handle NT service integration. This library facilitated the creation of NT services (daemons). Using the services library, three new components were created: *service_manager*, *process_manager*, and *proxy_client*. Of those three, process_manager and *proxy_client* were instrumental in creating a hybrid system. The service_manager was more of a utility component that closely mirrored the operation of NT's own Services applet.

Table C-1 Items Ported or Created on NT

	Libraries	Components
Ported	libthea libsys libenvelope libal	kined sensim simcon gen*
Created	service	service_manager process_manager proxy_client

C.3.2 Spread on NT

a. At the same time that Amalthea was being ported to Win32, investigations of the Spread group communications package on NT were undertaken.

b. Testing of Spread was performed in several steps, each subsequent step adding another variable into the configuration. At this stage, testing stressed compatibility and operational stability more than robustness or performance. Three utilities were used to test communications compatibility:

(1) A version of the Spread-supplied sample *user* program modified for multithreaded operation under NT
(2) A similarly modified version of the Spread-supplied sample *flooder*
(3) A custom program that sent *n* messages of *b* bytes, ported for both NT and Solaris

c. Numerous test cases were run under each of the following configurations, listed by purpose and order of complexity:

(1) **NT single daemon operation** – Multiple clients on a single NT box
(2) **NT daemon to NT daemon interaction** – Multiple clients on multiple NT boxes
(3) **NT daemon to Solaris daemon interaction** – Multiple clients on multiple NT boxes and multiple Solaris boxes

d. Spread was configured to use the same TCP/IP communication method as the current HiPer-D system, that is, multiple point-to-point connections. The first time an NT daemon was introduced into a network of Spread daemons running on Solaris, all of the other daemons crashed. This just turned out to be an interesting requirement of the Spread configuration files. Nodes had to appear in the same order in each configuration file. A likely assumption is that each Spread daemon determines its unique identifier according to its location within the configuration file. This problem was not experienced under Solaris because our Unix systems shared a common file system.

e. Other than that initial stumble, no errors were encountered through this stage of testing. Performance testing was deferred until tests could be conducted against the actual system. Additionally, the communications layers of Amalthea were ready to be linked with the Spread library, so focus was shifted to that task. One problem encountered was a program model inconsistency between Amalthea and Spread. Microsoft's compiler refused to link to Spread, giving errors related to *errno*. The primary suspicion was that Spread was built using single-threaded C libraries. (Recall Amalthea was compiled using multithreaded C libraries.) Dr. Yair Amir, the developer of Spread, confirmed this and built a version that utilized the multithreaded C libraries. Following this modification, compilation of the ported Amalthea libraries was complete.

C.4 Running the System

a. Although all of the SENSIM capsules had been ported, only two capsules, namely, KINED and SENSIM, were directly integrated into a running system. The remainder of the SENSIM modules were peripheral components and were not required in order to run the system. The various gen* capsules were manually started, so technically they did not have to be integrated. However, *gen_newt* was used often enough that a specific plan file was created for it.

b. Before running any of these capsules on NT, the corresponding Solaris capsules had to be disabled. This involved removing some of the capsule execution dependency definitions in the System Control plan files. Execution dependencies existed on KINED from KINED_BROKER, TRACK_CONTROLLER, and TRACK_GEN. Execution dependencies on SENSIM existed from TRACK_GEN. The plan files for those modules were modified to allow them to start without waiting on KINED or SENSIM. For now, the NT modules would be started by hand. Automatically starting NT modules through System Control was deferred until JHU/APL had a better understanding of the more basic integration details.

c. System performance would be measured in terms of track loads and track latencies. Track load refers to the number of tracks that the system is handling at any one time. A track report is generated for each track at some defined periodic interval. These reports formed the bulk of the network traffic sent throughout the system. For simplicity's sake, an interval of one report per second was used so that the track load corresponds on a one-to-one basis with the number of reports. The higher a track load the system could handle, the better. Previous tests on a full Unix system showed that it could handle track loads approaching 8000 tracks (at one report per second).

d. Track latency refers to the time interval between when a track report was sent by one capsule and when it was received by another. Lower latency numbers represent better performance. Latency measurements were taken at various points in the progression of a track, starting at KINED and ending at model_client. On a full Unix system, track latency between any two adjacent capsules averages under 30 ms. In the hybrid system, the interval between KINED and KINED_BROKER is of particular interest because it represents the first point of transition between NT and Unix.

e. The first tests were run on a uniprocessor Pentium 200-MHz machine with 64 M RAM (machine name *a2dwl*). Almost immediately the first interoperability issue surfaced; neither KINED nor SENSIM would join the RTDS_Signon group. The Spread-supplied USER program was used to monitor what, if any, messages where being sent to that group. By monitoring the RTDS_Signon group, the problem was traced to an unhandled endian conversion in rtds_if and tns_if. While these interfaces properly converted most messages, they did not convert zero-length messages. Although no actual endian conversion was required on zero-length messages, Spread reports endian differences and the AL layer issues an up-call to the conversion function. The conversion function did not properly handle this situation, resulting in the message being dropped rather than acknowledged.

f. After making the appropriate changes and recompiling the affected capsules, both KINED and SENSIM were able to join the required groups. In fact, the entire system seemed to be running. An ownship process could be started from Unix, tracks injected into the system through both Solaris and NT versions of gen_newt, and track flow observed from KINED through to the model client.

g. On the first run, the system showed extremely high (4 to 5 seconds or more) latency times, but it was soon realized that the NT and Solaris machines were not properly time synchronized. Since latency times are calculated using the local machine clock, an accurate reading can only be obtained if all machines in the system are properly synchronized. Time on the Solaris machines was maintained via Network Time Protocol (NTP) daemons on each Solaris box and a single time source. To synchronize the NT machine to the rest of the system, a suitable NTP service for NT was needed. Seven freeware and shareware NTP clients were obtained from the TUCOWS software repository (http://www.tucows.com/). Each client was examined in the areas of ease of use and simplicity. Many of the clients offered more functionality than necessary, such as time server capabilities. In the end, Dimension 4 by Think Man Software (http://www.thinkman.com/~thinkman/dimension4) was selected because of its simplicity, ease of use, and availability.

h. The NT system was synchronized to the same NTP server that the Solaris machines used. This resulted in latencies that were in the neighborhood of 200 ms as measured at KINED_BROKER. This was still an unacceptably high latency. It was suspected that the network connection of the NT machine, at the time a 10baseT switched connection, was the major limiting factor. All of the Solaris machines were running on shared 100baseT connections. In this configuration, the NT machine was only able to maintain approximately

1000 tracks before flooding its network connection. Further testing was postponed until a 100baseT connection could be established.

i. Following the connection of the 100-Mbit line to the NT machine, performance gains were immediately noticeable. Average track latencies dropped to 70 to 80 ms – still high compared to the Solaris machines, but acceptable for the time being. As the track load of the system was increased, strange results appeared in the average track latencies reported at each capsule. At just over 3000 tracks model_client would show stale track reports (latencies in excess of 500 ms). Because the only change introduced into the system was the addition of NT, the assumption is that one of those components, or perhaps NT itself, is causing a bottleneck. A perplexing aspect of this problem is that both KINED_BROKER and TRACK_PROCESSOR show track latencies around 100 ms. Therefore, between TRACK_PROCESSOR and model_client something is causing latencies to increase drastically. JHU/APL is still unsure what is causing this problem.

j. Ignoring the model_client latency mystery for the time being, JHU/APL continued to increase the track load. Unfortunately, as track loads neared 6000, CPU utilization in the 200-MHz Pentium approached 100%. JHU/APL decided to move testing to a more powerful Symmetric Multiprocessor (SMP) system with two Pentium II 300-MHz CPUs and 128 MB RAM (machine name *taurus*). A *switched* 100baseT connection was established and NTP time synchronization was installed. However, HiPer-D was unstable on this machine configuration. As tracks were injected into the system, the Spread daemon crashed with packet creation and packet delivery errors. The crashes seemed to occur at random points with no identifiable trigger events. It was not clear whether this was an issue with the SMP configuration, Spread, the NT capsules, the NT machine configuration, or some combination thereof. JHU/APL consulted Dr. Amir for assistance in exploring the possibility of an error in Spread. He built a modified daemon with increased debugging capabilities, but it was not possible to determine anything else about the problem through the use of this new daemon.

k. Because the error manifested itself only on this particular machine (*taurus*), we suspected a hardware or software configuration error or an SMP incompatibility. To isolate this suspicion, JHU/APL ran the hybrid system using another NT machine. Similar in configuration to *taurus*, the new machine was also an SMP but with only a single Pentium II 400-MHz, with 128 MB RAM (machine name *gemini*). Again a switched 100baseT connection was made and time synch software installed. On this machine, the Spread daemon did not exhibit any unexpected packet handling. Further investigation consisted of starting daemons on both *gemini* and *taurus*, then running the system from *gemini*. No HiPer-D components were executed on *taurus*, only the Spread daemon. Interestingly, the daemon running on *taurus* still crashed in this situation, indicating a problem with the underlying hardware or software installed on *taurus*. Following a clean reinstallation of NT, *taurus* no longer exhibited packet-handling problems.

l. Both *taurus* and *gemini* now appeared capable of running the hybrid system. However, an intermittent problem with KINED was experienced on both machines. At times, KINED stopped responding to the injection of new tracks. It would still send track updates for any tracks that were already in the system, but it would not start any new tracks. The error occurred at track levels ranging from no tracks (just starting) to hundreds of tracks before

locking. The normal solution was a shutdown and restart of the HiPer-D system. In the interest of time, investigations into the cause of the error were shifted into the background as other test and performance measurements were continued.

m. Analysis of all running HiPer-D threads revealed two threads that consumed the majority of all system resources. One was the single Spread daemon thread, the other was a KINED thread that listened for new Spread messages. On the first machine (a2dwl), each of these two threads accounted for nearly 50% of all processor utilization at 6000 tracks, effectively inundating the entire machine. As expected, the faster Pentium II machines were much more capable of handling the CPU demands of the system. At 6000 tracks, processor utilization was approximately 35% on *gemini*, and approximately 20% per processor on *taurus*. Network utilization averaged 1.5% per 1000 tracks. It certainly seemed like these machines and connections could handle the resource requirements that HiPer-D put forth. However, there were still high latencies in certain parts of the system. Again, KINED_BROKER and TRACK_PROCESSOR maintained relatively low latencies while model_client still reported much higher numbers. Performance tuning was delayed reliability and Unix interface issues could b resolved.

C.5 Integration with System Control

a. At this point the hybrid system was stable enough to include as part of Demo 98. As with the Unix-only configuration, JHU/APL wanted to be able to start up the entire system with a single command to system_control. Until now, all of the NT capsules had been started by hand. Therefore, the first step in an automated startup was an interface module between Unix system_control and the NT capsules. One option was to port System Control's node_manager and agent programs to Win32. System Control could then communicate directly with the ported modules. However, node_manager and agent utilized platform-specific features such as forking and parent-child relationships, which extended beyond the capabilities of the Amalthea library. This made this option neither practical nor feasible given the time constraints. A more applicable solution involved the integration of System Control on the Unix side and a separate program called process_manager on the NT side.

b. Process_manager was created as an NT system service that allowed the remote invocation, monitoring, and termination of an application. Commands and requests can be sent to process_manager through a regular socket connection. A single process_manager host can service multiple client connections, allowing the remote invocation of multiple applications on a machine (see Figure C-1).

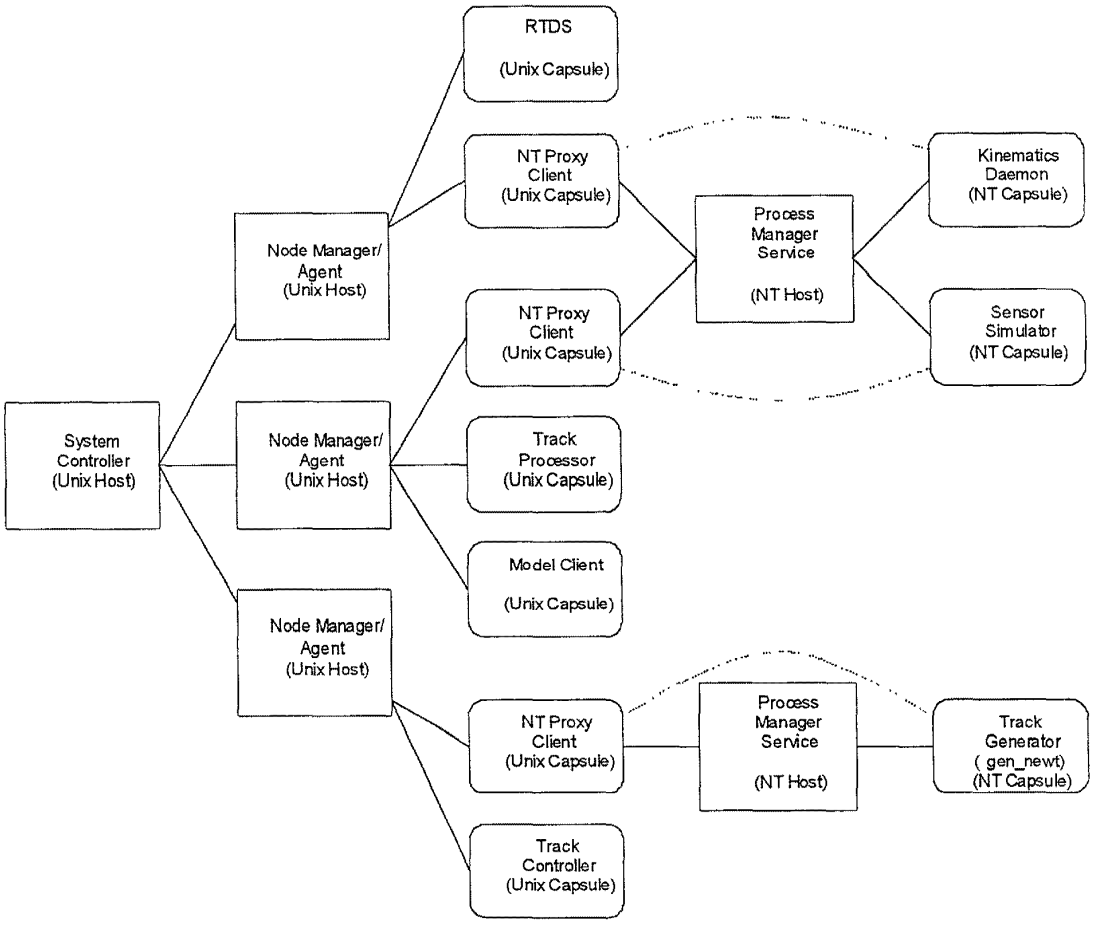


Figure C-1 Hybrid System Configuration

c. A new Unix program called *proxy_client* was developed. *Proxy_client* runs as a normal System Control client and connects to process_manager through a socket. Through command line and environment variables, JHU/APL could specify what application to run, along with an initial environment. *Proxy_client* can be started from any machine, local or remote as long as it can make a socket connection to an NT host running process_manager. Each *proxy_client* is bound to exactly one application. In addition, each *proxy_client* maintains a heartbeat between itself and the application to which it is bound. If the application exits, *proxy_client* will also exit. Similarly, if *proxy_client* exits, the application it started will exit as well. If the host process_manager service somehow dies, all *proxy_client*s and applications tied to that host will exit.

d. A version of *proxy_client* was developed for NT during the Amalthea Unix-to-NT porting process. It was used as the starting point for the creation of a Solaris version of *proxy_client*. Fortunately, the NT version of *proxy_client* was written using the Amalthea libraries, facilitating a port to Solaris. However, creating a Solaris version involved more than a

simple recompile of *proxy_client* because a number of Amalthea packages were added or modified to implement the necessary features (see Table C-2).

e. Porting the affected packages from Win32 back to Solaris was a straightforward process. Again compiler #define directives had been used to separate platform-specific code. As a result, simply overwriting the relevant Unix versions with the new NT versions was all that was required to synchronize the affected modules.

Table C-2 Amalthea Packages Ported Back to Unix

Package	Modification
thea_socket	Support for Unix-style sockets
thea_process	Support for additional process functions
thea_log	Support for logging of system errors
thea_string	Support for additional string functions
thea_assert	Added

f. After updating the Amalthea libraries, *proxy_client* compiled on Solaris without any modification. However, changes were needed to handle platform-specific characteristics in the user environment. The Win32 version of *proxy_client* was designed to forward all environment variables to the application it starts, but it does not make any sense to pass a complete Unix environment to an NT application. For the purposes of this phase, JHU/APL restricted the forwarding of user information to only those variables that applied to the Windows environment (see Table C-3). Three new variables were introduced under Unix to specify an NT home directory (WIN_HOME), NT account username (WIN_USER), and NT account password (WIN_PASSWORD). In general, these variables were not used; however, they were added to parallel the functionality of the original NT version of *proxy_client*. Without any modification, *proxy_client* would try to authenticate account information on NT using the Unix account. This would work if an account exists under NT with the same username, but this is both impractical and unnecessary. Process_manger did not require authentication to start an application.

Table C-3 Environment Variables Forwarded by *proxy_client* (Unix)

Environment Variable	Values	Purpose
LOGGING	YES | NO	Enable / disable logging of output
CONSOLE	YES | NO	Enable / disable console interaction (should be ON)
CAPSULE	string	Specify the capsule name
STDIN	string	Specify and new STDIN source
STDOUT	string	Specify a new STDOUT destination
STDERR	string	Specify a new STDERR destination
WIN_HOME	NT path	NT application home directory; replaces HOME
WIN_USER	string	NT account username; replaces USER
WIN_PASSWORD	string	NT account password; replaces PASSWORD

g. The Solaris *proxy_client* was able to start, monitor, and stop an application remotely on an NT host. JHU/APL now had all the necessary components to implement an automatic

startup. HiPer-D would be configured just as it was on the Unix-only system, with the exception that KINED and SENSIM would be running on NT.

h. To start the NT versions of KINED and SENSIM automatically, system_control was used to start two instances of *proxy_client*. Plan files were created for two *proxy_clients* as system_control capsules. One proxy_client would be bound to KINED, the other to SENSIM. Each client was responsible for starting and monitoring its corresponding application. In this manner, system_control was able to start, stop, and monitor the NT capsules indirectly.

i. The results were quite impressive. The hybrid system ran just as it did in a pure Unix configuration. All of the functionality from system_control, such as the ability to include capsule dependencies and restart information, was available to the NT capsules. Once a proper configuration had been installed on an NT host – that is, process_manager and Spread were both running – no further interaction with the NT machine was necessary.

C.6 Current Problems and Issues

a. JHU/APL has recently moved from Spread 3.08 to Spread 3.09, which was released in October 1998. No API changes were made in the new version. The major differences were the inclusion of a multithreaded library (similar to the custom-built library from 3.08) and support for IP-Multicast to a specific network segment.

b. Two problems plagued the system. The first is a seemingly random locking that KINED occasionally experiences. Since moving to Spread 3.09 this problem has not been observed, but more extensive testing is required before a conclusive statement can be made about this error. The second, and more intriguing, issue was a repeatable view change error. The problem is as follows: when gen_newt is run on the same machine as KINED and SENSIM, KINED will report a view change from the unknown group /RefGrp/SIM_Domains, as well as duplicate ownship messages from an external source. After some investigation, it was determined that KINED never joins the /RefGrp/SIM_Domains group (it is not supposed to), yet it still receives view change messages for that group. To confirm this, two instances of the user program were started. The first was used to simulate the group membership of KINED; the second would simulate the actions of gen_newt. An interesting observation was made: the KINED simulation would receive invalid view changes as soon as the gen_newt simulation was started! Furthermore the problem only occurred when gen_newt was started with a private name that began with a capital 'K' or earlier (i.e., A, B, J, 3, 4, etc.). Another way of stating this is that KINED would receive bad view changes whenever gen_newt connected with a private name whose first character was ASCII value 75 (decimal) or lower. This conclusion was made after repeated tests with different private names for gen_newt. By default, gen_newt used the private name GEN_NEWT when connecting to the Spread daemon. When the connection name was changed from GEN_NEWT to GEN_NEWT and the entire HiPer-D system was rerun, no errant behavior was observed.

c. Through deeper investigations into the view change error, the problem was traced back to an incorrect compiler optimization of the Spread daemon by Microsoft Visual C++ Version 5.0. Dr. Amir has built a new version of the daemon without any optimizations. Thus

far, no strange view change behavior has been observed, but testing will continue to determine if any other problems remain unresolved.

C.7 Future Goals

a. A major goal for the near future is a migration of the NT versions of Amalthea and the various SENSIM applications back to the single code base that supports the Unix systems. Since the NT version of Amalthea was developed, a number of modules on the Unix side have undergone major revisions. Message structures and general behavior remain similar for the time being, but JHU/APL cannot guarantee how long this will be the case. It is quite obvious that a dual code base will not only be difficult to maintain, but may also introduce additional problems into the system. For the majority of the modules within Amalthea, reintegration into a single code base will not be that difficult. Compiler #define directives were heavily used to separate platform specific code. In addition, no Unix areas were modified when NT code was added. If no changes have been made to the Unix code, all that will be needed is to copy the NT version over the Unix version. If changes have been made, however, those changes must be incorporated into the NT versions, and possibly some of the NT-specific areas must be rewritten to implement algorithmic changes.

b. The last major area of investigation deals with improving performance. Among other things, JHU/APL will address issues such as whether an SMP system significantly improves throughput, or whether the physical location of machines relative to each other has any significant effect on latency times.

C.8 Lessons Learned

a. Over the course of this investigation JHU/APL came into contact with many aspects of Windows NT 4.0. Just like every other system, NT has both its advantages and disadvantages. These qualities need to be identified, and in some cases modified, if possible. The following are observations regarding the experience of working with NT.

b. Setting up a new machine for NT is a straightforward process. Machines that supported booting from compact disk, read-only memory (CD-ROM) were the easiest to set up: Simply insert the NT CD into the CD-ROM drive and power up the system; the installation program starts automatically. Systems without bootable CD-ROMs required the use of a boot disk, which, although slower, was not any more complicated. Hardware detection and compatibility, a common issue in both the Unix and NT domains, was not a problem in this situation. NT was able to detect all system-level components such as the peripheral component interconnect (PCI) bridging hardware, small computer system interface (SCSI) controllers, and Ethernet adapters. Certain subsystem components, including video and sound cards, defaulted to a generic driver. However, an area in which NT has an obvious advantage over Unix is extensive vendor support for the NT platform. In every case, new device drivers for any nonnative devices could be obtained from the manufacturer's website. It is almost guaranteed that any new devices, both mainstream commercial and more industry specific, will support operation under NT.

c. Industry support is also prevalent in the area of application software. Microsoft's Windows 9x operating system has dominated the consumer and business markets. Windows 9x runs a subset of the Win32 API, while NT utilizes the complete Win32 specification. As a result, nearly all applications that run on Windows 9x will run on NT with little or no modification. Some vendors take advantage of the added functionality in the NT implementation of Win32 to make their applications even more robust under NT.

d. A major issue with the state of software deployment on NT is the intentional or inadvertent modification to the system through the installation of a user application. Microsoft software often causes this problem. For instance, the installation of Microsoft Internet Explorer 4.0 or the Microsoft Office Suite – both user applications in function – added new files to the NT system directory. What's more, these applications often modify critical system components and dynamic link libraries (DLLs), sometimes going as far as to modify the kernel. The installation of one or more select applications, and the subsequent system modifications these applications made, caused the original packet instabilities on *taurus*. It has not been determined which particular application or applications are responsible. In setting up NT for running HiPer-D the NT system was kept as "clean" as possible; only those things that were absolutely necessary were installed. The intent here is not to advocate running a bare system – that would defeat the purpose of a computer. However, care should be taken when adding any application to the system. One option is the use of some sort of monitoring program, such as any of the commercial uninstall programs, that is capable of recording the state of the system before and after an application is installed. These utilities can track changes to any files, new files that were added, and modifications to the registry. In this manner, system administrators have direct knowledge of exactly what has been changed and whether the installation of a particular application may cause instabilities.

e. Growing support of NT in the software vendor community throughout the years has prompted the creation of many development tools. Consequently, NT has one of the most feature-rich development environments of any platform. Microsoft's Visual Studio Integrated Development Environment (IDE), if utilized to its full extent, is extremely versatile. It tracks multiple sources, generates dependencies, and provides useful variable browsing capabilities, among other things. In terms of debugging, Visual Studio allows source-level debugging, attaching to an already running process, debugging individual threads, and a multitude of other features. Other vendors have also created various development tools, such as SoftIce debugger and Wdiff visual diff utility. Most of our executables were generated through individual makefiles and the command line compiler rather than through the IDE. Because of this, advanced development features such as automatic dependency generation and incremental building were unavailable. As a result, modifications to any of the libraries required a complete rebuild instead of a selective relink, slightly increasing time requirements. Fortunately, the source level debugger could still be used to trace the operation of the NT capsules.

f. NT is flexible enough to simulate many of the Posix-derived functions that Amalthea exposes. Most required functions had some sort of corresponding Win32 call; the functionality of those that did not could be simulated through a combination of calls. The more subtle effects of simulating Unix in this manner have yet to be determined. Possible implications include improper operation, hindered performance, and reduced stability. However, all tests thus far

have not shown any errant operation or loss of stability that can be attributed to the porting of Unix calls. Win32 was never designed to simulate Unix calls; it is a different programming model that supports its own concepts and operations, including access to the characteristic Windows Graphical User Interface (GUI) routines. An interesting investigation might be the creation of graphical front-ends to the various HiPer-D components, possibly allowing dynamic interaction and tweaking of the modules, as well as improved presentation and reporting capabilities.

g. Perhaps the most valuable lesson learned from this experience is that Windows NT is a viable operating system that should not be overlooked. Not only was NT capable of running HiPer-D applications, but it is able to achieve relatively good performance and reliability from those applications without an extensive amount of tweaking. Just looking at latencies and track loads will show that HiPer-D running on Unix has a clear-cut lead in terms of performance. However, keep in mind that while the Unix configuration has been tweaked and modified for years, the migration to NT has occurred in just over 4 months.

APPENDIX D

HISTORY
OF
HIPER-D TRACK CORRELATOR AND FILTERING PROCESS

The Johns Hopkins University
Applied Physics Laboratory
Laurel, Maryland 20723-6099

APPENDIX D

HISTORY OF HIPER-D TRACK CORRELATOR AND FILTERING PROCESS

The Johns Hopkins University
Applied Physics Laboratory
Laurel, Maryland 20723-6099

D.1 HiPer-D Correlation and Tracking (HCT)

a. The earliest HiPer-D work was focused on the ability to move existing tactical capabilities into a networked, distributed computing environment. HCT was an effort to provide a network-distributed implementation of the CEP correlation and tracking capabilities. This actually consisted of three separate functions (Track Management, Gridlock, and Update) as depicted in Figure D-1. There were two major objectives for this phase:

(1) portability
(2) communications development.

b. The CEP code was written in C and implemented on Motorola 680x0 Versa Module Eurocard (VME) boards with the pSOS executive and a set of JHU/APL-developed messaging services. The clean, well-defined interfaces among the three application functions (Gridlock, Track Management, and Update) provided an obvious strategy for the port to HiPer-D. Network-based messaging services were developed to run both on standard network (Ethernet) environments, as well as Mach operating system native messaging;[1] and a surrogate for the pSOS executive was constructed to run on the host operating system (either Mach or Unix).

c. The porting efforts were quite successful. At the time of the I1 demonstration, HCT ran on Ethernet-networked PCs with Mach, Ethernet-networked Digital Decstation 5000s under Digital Unix, and on an Intel MPP machine called the Paragon using Mach and an internal high-speed switch network.

d. The communications results were mixed. While the Paragon high performance switch network was effectively able to negate the performance distinction between local and remote communications, the situation in the Mach-based PCs was different. With buffering set for tolerable latencies, the PC was still able to achieve only ~1000 messages/sec to a remote process vs. over 10,000 messages/sec to a local process.

[1] The Mach operating system, at the time, was a promising research endeavor that emphasized distributed computing and provided interprocess messaging services that allowed the processes to be placed among a collection of machines transparently.

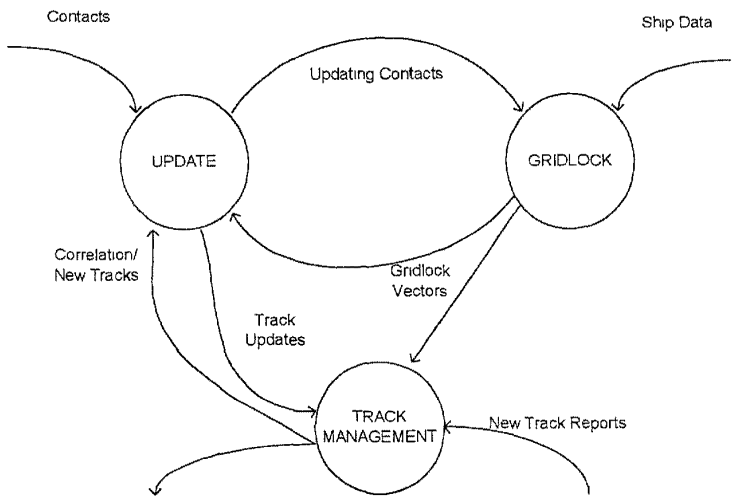


Figure D-1 CEP Tracking and Correlation

D.2 Track Correlation and Filter (TCF)

a. The change from the HCT to TCF (Track Correlation and Filter) was motivated by the desire to address functionality more directly related to Aegis. Basically, the external interfaces initially established for HCT were maintained, but the HCT was completely replaced with new correlation and tracking that was derived from the Aegis C&D specification. Aside from this shift of focus, the other objectives were to investigate scalability and fault tolerance. A major component of this shift was also a change in messaging, away from simple Mach and network messaging services to a higher level set of services (the ISIS Distributed Toolkit) that offered delivery and ordering guarantees that are helpful in constructing scalable and fault-tolerant software components.

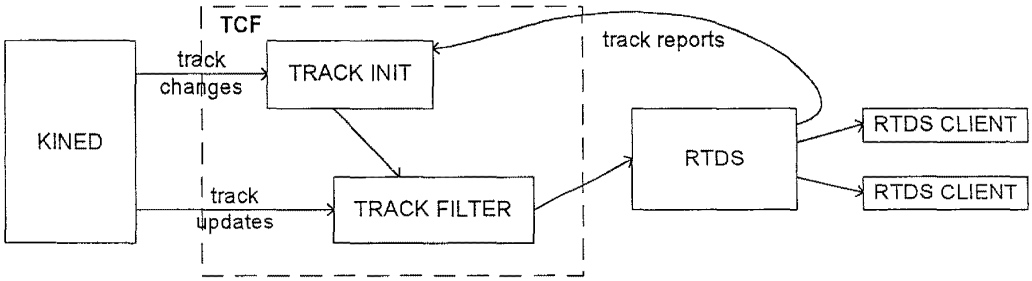


Figure D-2 HiPer-D Track Correlation and Filtering

b. TCF consisted of two modules – Track Init and Track Filter (see Figure D-2). The Track Init module performed correlation for a new track and entered the track into the system. The Track Filter implemented an alpha beta filter for maintaining state on each track. An "alpha beta" filter smooths a set of periodic measurements of some value by means of the equation:

NewState = Alpha * M + Beta * Last

Where:

NewState is the current estimate of the variable.
M is an X or Y position measurement.
Last is the previous estimate of the variable.
The alpha and beta coefficients should sum to one.
For alpha=1 beta=0, no filtering is performed. As beta is increased from zero to one, the effect of the filter is increased.

Track Filter implemented a primary/backup form of fault tolerance, but this was not thoroughly tested. Track Init had no form of fault tolerance.

c. A simulator (KINED) was created to model multiple sensors and drive the system with input data. Two separate communications groups[2] were created for delivering track data from the KINED to the TCF. Track change messages (new and lost) were routed into the Track Init, correlated, and forwarded to the Track Filter. Track updates were delivered directly to the Track Filter from KINED, bypassing the Track Init module. Track Filter then sent both the track change and track update messages to the RTDS (Radar Track Distribution Server), allowing all messages to be placed in one ordered stream. The separation of the communications groups providing input to TCF allowed for differing delivery characteristics – minimal latency is desirable for track change messages, but high throughput is more important for the track updates. Track change messages probably account for less than 1% of the total traffic in a running system.

d. AutoSpecial handling[3] created the need for new and lost track notifications with minimal latency (see Figure D-3). The sensor (simulated by KINED) determined whether a new track fell within an AutoSpecial region. In the event that one did, the sensor issued an *AutoSpecial tentative* new track message to Track Init. A communications group existed for clients interested in AutoSpecial events. Track Init would react to an AutoSpecial Tentative message by immediately issuing a tentative AutoSpecial message into the AutoSpecial group. The sensor was responsible for issuing a positive or negative resolution message (*AutoSpecial report*) shortly after issuing the tentative message.[4] If the resolution was positive, Track Init entered a new track, and RTDS clients would begin to receive reports on the new track.

---

[2] Process group communications are described in Section **Error! Reference source not found.**.

[3] AutoSpecial is a term used to describe a high interest or high threat track. Typically, some region around a ship would be defined, and any tracks present in this region would be classified as AutoSpecial. Weapons control systems would want to receive notice for any detected AutoSpecial tracks as early as possible.

[4] If this confirmation did not arrive within a prescribed time frame, a negative resolution would automatically be issued by Track Init into the AutoSpecial communications group.

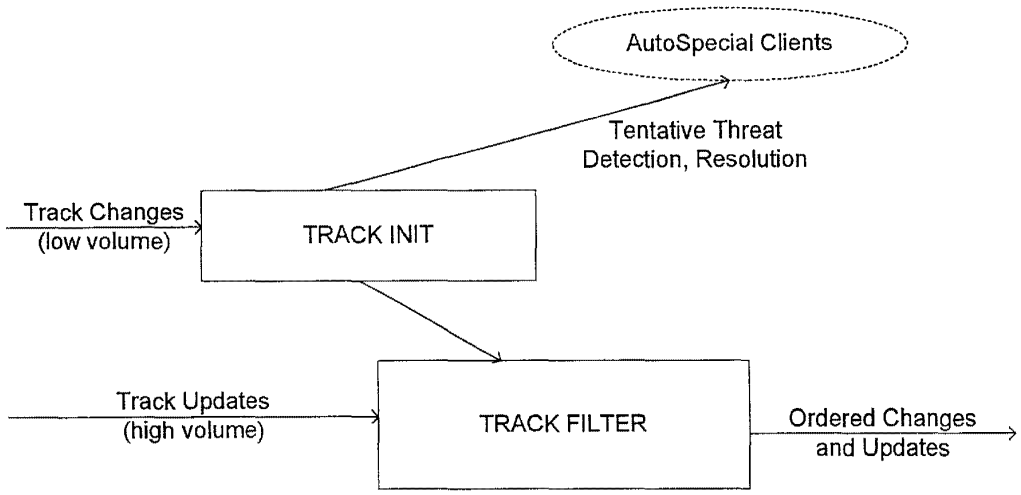


Figure D-3 TCF Information Flow

e. The Track Init module performed track correlation between sensors. This required that Track Init maintain a current state for all tracks in the system. However, track updates were only being delivered to the Track Filter. From Track Filter, they were forwarded to the RTDS. To obtain track data, Track Init actually became a client of the RTDS in order to receive recent state on all tracks (see Figure). This allowed Track Init to evaluate and compare any new tracks to the set of existing, composite/filtered tracks being reported by all sensors.

f. The Track Filter module combined the updates from multiple sensors and applied a simple filter to produce a composite track picture. This capsule could be replicated for fault tolerance, but filtering was performed by every Track Filter for every track. This simply means that the full load had to be carried by each Track Filter.

APPENDIX E

GROUP COMMUNICATIONS

The Johns Hopkins University
Applied Physics Laboratory
Laurel, Maryland 20323-6099

APPENDIX E

GROUP COMMUNICATIONS

The Johns Hopkins University
Applied Physics Laboratory
Laurel, Maryland 20323-6099

E.1 Introduction

a. One of the core technologies on which HiPer-D is based is process group communications. Process group communications provide a mechanism in which applications become members of a communications group. When a member sends a message in a group, all other members of the group receive the message. In this respect, process group communications are analogous to multicast communications. Process group communications extend the concept of multicast by providing reliable communications, by guaranteeing different levels of message ordering, and by providing operations associated with membership changes in the group.

b. Since its beginning, HiPer-D has used a process group communications package known as ISIS. ISIS provided the following characteristics:

(1) Reliable communications
(2) Levels of ordering including no ordering, first in first out (FIFO) ordering, causal ordering and total ordering
(3) Group membership change (referred to as group view changes or view changes) notifications that are synchronized in the message flow
(4) A state transfer approach that holds up the message flow to all members and allows a synchronized state transfer to take place among group members when a view change occurs.

c. Process group communications have been used throughout HiPer-D as the primary means for communicating among distributed processes. In general, the primary features that have been used are the reliable communications, message ordering, and group membership change notification. For example, the ISIS state transfer facility has been used to implement the fault-tolerant and scalable RTDS.

d. In 1997, STRATUS, the vendor of ISIS, announced that it would stop selling ISIS and that there would be no replacement product. This prompted a search for a new middleware product. It quickly became obvious that there were no commercial process group communications packages available. There were, however, several research packages available.

e. The research packages that were considered included HORUS, ENSEMBLE, Spread and TOTEM. HORUS is from the continuation of the research efforts that produced ISIS. ENSEMBLE is the latest product from the same research group. Spread is a second-generation product that is derived from TOTEM. HORUS and ENSEMBLE were not selected due to maturity and stability concerns. TOTEM and Spread appeared to have the stability required for use in HiPer-D. Spread was chosen because it is more advanced than TOTEM and because local support was available from the developer of Spread, Dr. Yair Amir at The Johns Hopkins University Homewood campus. All of the communications packages reviewed, including Spread, supported reliable communications, ordered messaging, and group membership services. None supported synchronized state transfer, which is a major component of the RTDS design. This meant that synchronized state transfer would have to be implemented in a layer above Spread and below the application.

f. JHU/APL's original approach to communications for HiPer-D involved the development of three thin layers of software between the application and the underlying support mechanism. These layers are known collectively as Amalthea Communications or "Amalthea comms". The initial research into a replacement for ISIS quickly led to the realization that the Amalthea comms components that supported process group communications would have to change significantly if the move away from ISIS was to be accomplished without redeveloping HiPer-D applications.

E.2 Amalthea Communications Layers

a. Amalthea Communications consists of three layers: domain classes, domain class support (DCS), and group management services (GMS). Amalthea communications is written in ANSI C. Amalthea communications layers are shown in Figure E-1 and are explained in subsequent paragraphs.

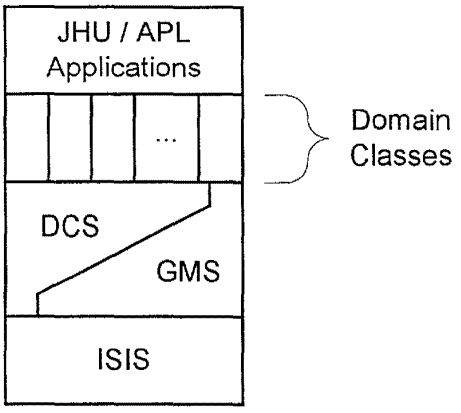


Figure E-1 Amalthea Communications Layers

b. Domain classes are actually part of the application in that they are defined by the application. Domain classes contain message definitions and the definitions of all the routines that the application would use to send and receive messages. The idea is that this information is

defined once and is placed in a domain class library. Any application that needs to send or receive messages using that domain class would link with that library. This ensures that all senders and receivers of messages defined in a domain class have the same message definitions, thus preventing problems associated with senders and receivers operating with different revisions of a message structure.

c. Domain classes support the idea of multiple instances of a domain class. Applications have the ability to define a domain class and then to use that same domain class to create a separate instance of the domain class. Messages sent in one instance are not received in a separate instance. An example of the use of multiple instances is a server's communication with multiple clients. The message structures and access routines would be the same for all clients but each client would need to receive a different set of messages. The server would create a separate instance of the domain class for each client. In this way, each client would receive only those messages specifically destined for that client, and the server would be able to use the same code to talk to all clients.

d. Domain classes do not have to be ISIS-based. The concept can be used to implement a communications path using any underlying communications protocol. In the 1998 Advanced TCF efforts, a domain class was formed that used IP multicast as the underlying communication layer.

e. The domain class support layer contains support utilities used by domain classes. These utilities include mechanisms for sending and receiving blocks of data. Domain classes form a message and then send a pointer to a DCS send function. The DCS send function actually transmits the message via ISIS. Similarly, messages received from ISIS are forwarded to the appropriate domain class by DCS. Prior to 1998, DCS was the layer that isolated HiPer-D from changes in ISIS.

f. Group management services provide domain classes with special services associated with group membership changes. These services include the ability to join a specific group and to register a callback routine that will be called when a view change (membership change) message is received. An application uses the join call to register a callback routine that will be called when a data message is received, and a separate routine that is called if the sending machine has a different architecture than the receiving machine. This last callback is known as a conversion up-call. Conversion up-calls give the domain class the opportunity to perform byte swapping in situations where the sending machine has a different endian ordering than the receiving machine.

g. Other GMS services aid in the management of communications groups that contain both servers and clients and that use state transfer to implement fault tolerance. Before these services are explained, the basic approach of using ISIS synchronized state transfer must be explained.

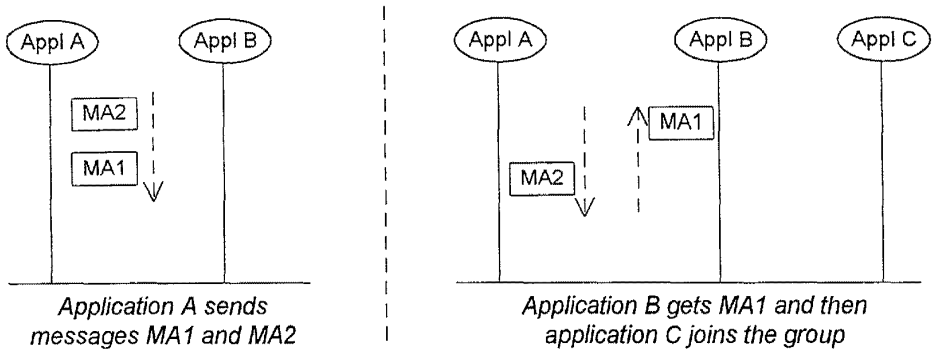


Figure E-2 New Process Joins Group

h. The left side of Figure E-2 shows Application A sending two messages, MA1 followed by MA2. The right side of Figure E-2 shows Application B receiving MA1 followed by the joining of Application C. When C joins, a group view event is started. At this point, ISIS completes the delivery of all messages that were sent prior to the view change even though the join attempt started before Appl B received MA2. This preserves consistent ordering of events among members of a group, even though actual delivery times may vary somewhat in a real-time sense. In this example, messages MA1 and MA2 are delivered. The left side of Figure E-3 shows Application B reacting to MA1 and MA2 by creating MB1 and MB2.

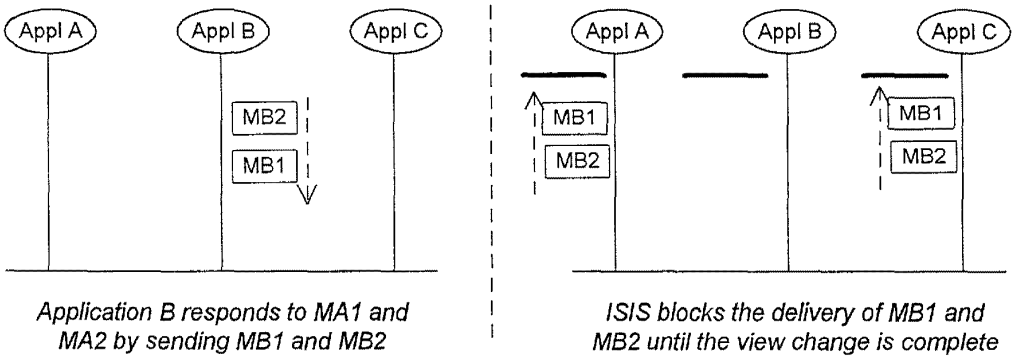


Figure E-3 Messages in Old View are Delivered

i. MB1 and MB2 are sent in the new view that includes Application C. ISIS blocks the delivery of MB1 and MB2 until the view change process is complete. This is shown in the right side of Figure E-3. After all messages from the previous view have been sent, ISIS provides all participants in the group with the opportunity to transfer state to the new joining member. GMS filters this and forwards the state transfer option to only the oldest member of the group. In this example, assume that Application A is the oldest group member.

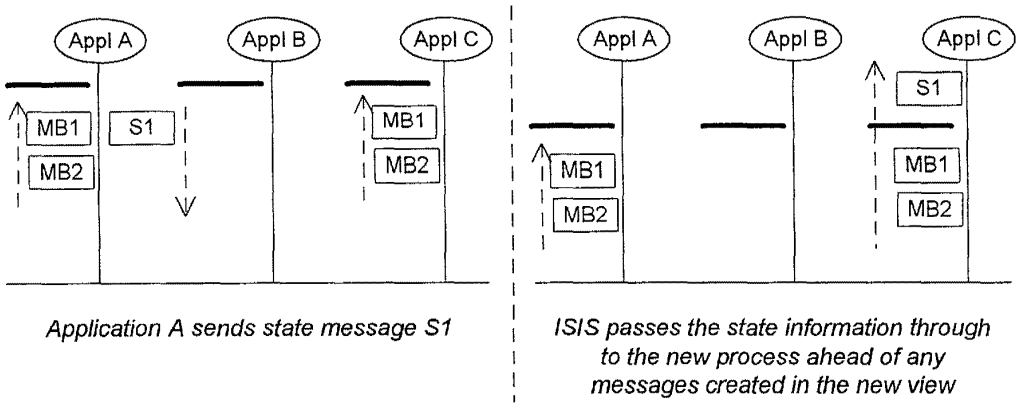


Figure E-4 Sate is Transferred

j. Figure E-4 shows Application A sending a state message in response to the delivery of a view change event from ISIS. The state information includes all information that the new member, Application C, will need to become a fully functional member of the group. In this example, that might include the fact that MA1 and MA2 were sent. This will prepare Application C for the arrival of MB1 and MB2. Once all state transfer messages have been sent, ISIS resumes the message flows illustrated in Figure E-5. At this point application C has been fully entered into the group, and the view change is complete.

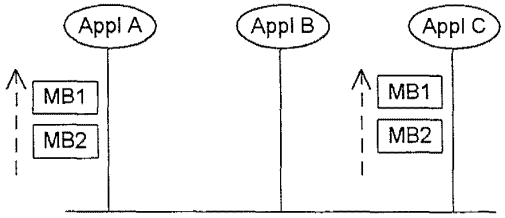


Figure E-5 Message Flow Resumes

k. Figures E-2 through E-5 illustrate the situation where a new process is added to a group. When a process leaves the group, either intentionally or as the result of a failure, a similar process occurs. All messages in the old view are delivered, assuming that there are receivers still left in the group. ISIS then holds any messages created in a new view and signals for a state transfer. GMS then selects the oldest member to perform the state transfer. When the state transfer is complete, ISIS resumes the message flow. The state transfer mechanisms illustrated here form the basic building blocks that JHU/APL used to develop fault-tolerant servers for HiPer-D.[1]

l. The examples above illustrate the basic case where all members of a group are of the same type. This situation is shown graphically in Figure E-6.

[1] Note that the state transfer mechanism is only implemented for applications that are capable of sending or receiving state transfer. GMS's use of the capabilities provided by ISIS is such that there is no performance penalty for applications that do not use state transfer.

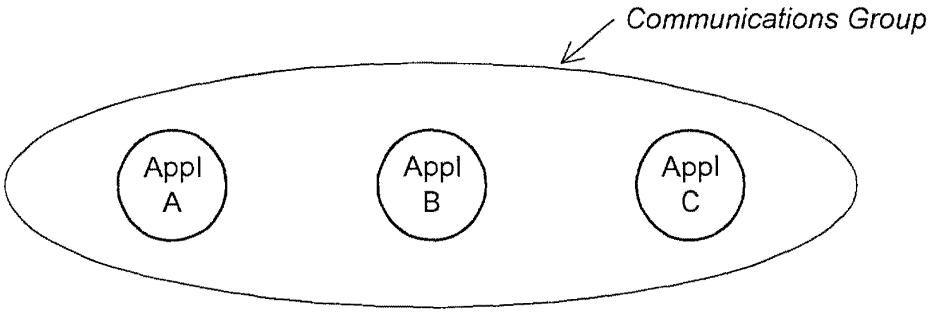


Figure E-6 Homogenous Applications Sharing a Communications Group

m. The situation is more complex when the fault-tolerant entity is a server. In this case servers must belong to a communications group in which they can communicate with each other and they must also belong to a group in which they can communicate with clients. The groups must be separate since clearly it would be undesirable for clients to be aware of or involved in communications among the servers. Figure E-7 illustrates this situation.

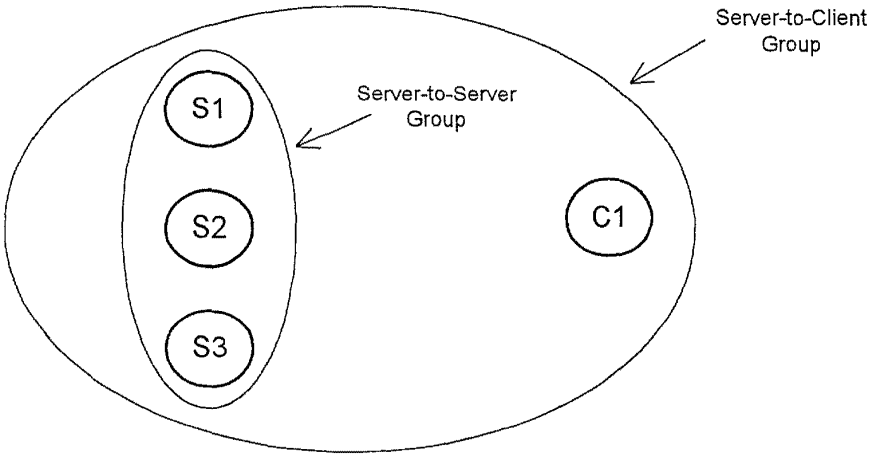


Figure E-7 Client / Server Using Group Communications

n. The situation where a server belongs to multiple groups complicates state-transfer. ISIS uses the synchronized state transfer process when any process joins a group. Clearly, this would be inappropriate when a client in Figure E-7 joins the Server-to-Client group because state transferred would be server state information. GMS supports the concept of a reference group to assist in this situation.

o. A reference group is a group in which applications of the same type communicate. The Server-to-Server group in Figure E-7 is a reference group. GMS implements the concept of reference groups with the following rules:

(1) Only processes of the same type may be in a reference group.

(2) State transfer only occurs between processes in the reference group.
(3) A Process may only belong to a single reference group.
(4) Any process using state transfer must belong to a reference group

p. GMS also supports the concept of fully qualified members of a group. A group member is fully qualified if it has completed all of its initialization activities, including joining all appropriate groups, and is ready to participate in the group. Scalable servers use this capability to determine when a new server is ready to begin processing.

q. GMS supports the concept of a synchronized group. A group is synchronized if all of its members are fully qualified. A group is unsynchronized if any of its members are in the process of becoming fully qualified or if any of its members have departed and the view change has not completed. GMS provides optional up-calls to signal changes in synchronization. Fault-tolerant servers use this to maintain consistency during view changes.

E.3 The Adaptation Layer Approach

a. In considering the impact of removing ISIS from the system, it was realized that the effort would impact both DCS and GMS and to a small degree the domain classes. This would be a high-risk process since all of JHU/APL's HiPer-D components relied heavily on these layers. In order to prevent this from happening again in the future, it was decided that a new Adaptation Layer (AL) interface would be created. This is shown in Figure E-8.

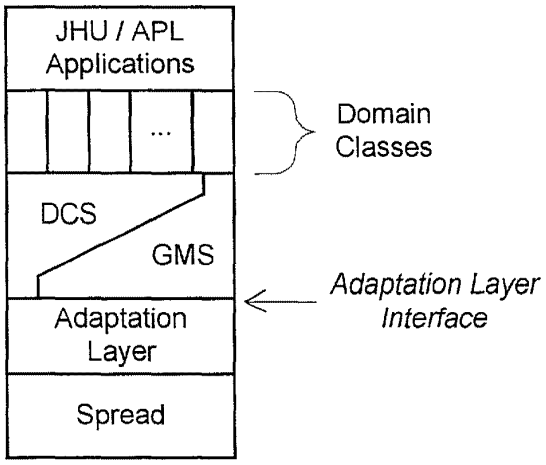


Figure E-8 The Adaptation Layer

b. The Adaptation Layer Application Programming Interface (AL API) identifies a basic set of group communications capabilities on which DCS and GMS can rely.[2] If the underlying process group communications package supports those capabilities then the AL would be thin and would pass calls on to the underlying implementation. Any capabilities specified in the AL

[2] Note that in the following discussions the term "Application" refers to the DCS / GMS layers because they are the users of the AL. The users of Amalthea comms, which are HiPer-D applications, never directly reference the AL.

API that were not available in the underlying process group communications package would be implemented in the AL.

E.4 Spread

a. Spread implements process group communications by using daemons that are placed on different processors throughout the system. Applications communicate with Spread through a linked library that establishes a connection with a daemon. All communications with other Spread applications are routed through the daemons and up through the library to the application. Spread allows there to be any number of daemons in a system. An application may connect with a daemon on the same machine on which it is running or it may connect to a daemon on a remote machine. To improve fault tolerance, HiPer-D places a daemon on each machine and requires that all applications on a machine connect with the local daemon. This is illustrated in Figure E-9.

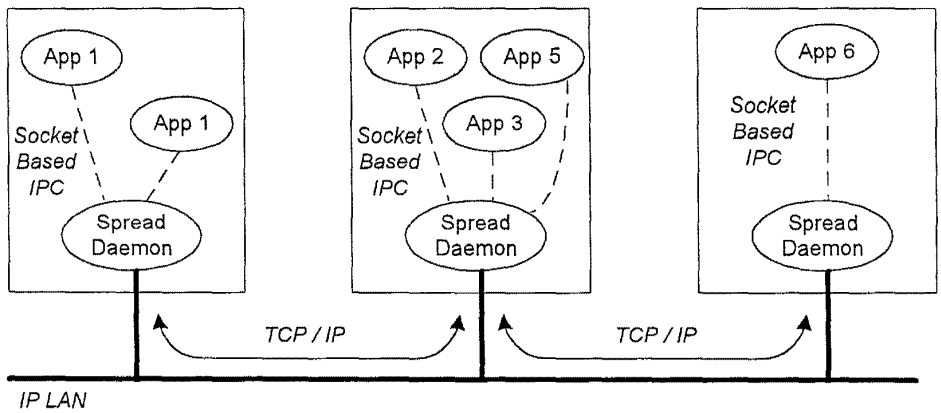


Figure E-9 HiPer-D's Use of Spread

b. Spread daemons may be configured to communicate using either IP multicast or TCP/IP. To date TCP/IP has been used to minimize any potential problems associated with the ATM network in use at NSWCDD.

c. Spread's programming interface is simple compared to ISIS. It consists of a connect call, a receive call, a pair of send calls, a join call, a leave function, and a handful of utility functions.

d. The connect function is used to establish a connection between the application and the Spread daemon. The dashed lines in Figure E-9 illustrate this connection. This connection forms an endpoint to which Spread assigns a name that is unique within the system. Because

there is one endpoint per application, the endpoint name can be thought of as Spread's name for the application.[3]

e. The receive call is the only function through which Spread returns group information to the application. Messages from other members of the group are returned from this call as well as special Spread-created view change messages. When application messages are returned, the name of the endpoint that sent the message is also returned. The receive call also returns a flag if the machine that sent the message has a different endian architecture than the receiving machine. This warns the user that the data in the message buffer may need to go through a byte swap operation. Each view change message identifies the reason that the change occurred and contains a list of the endpoints that are in the new view.

f. Spread supports a send call that allows the user to pass the address of a data buffer and its size to the call. To use this call the user must marshal the message into a contiguous space in memory. Spread also supports a scatter send call that allows the user to forward a list of pointers and sizes to Spread. This allows the application to send a message that is made of several components without having to first copy them into a contiguous buffer space.

g. The Spread join call is used to join a group and the leave call is used to leave a group. There is no limit to the number of groups that an endpoint can join. All messages from all groups that an endpoint has joined are received through the receive function call. The receive call returns the name of the group in which the message was sent along with the message.

E.5 Adaptation Layer API

a. Those capabilities that are provided by currently available process group communications packages have influenced the specification of the AL API. ISIS implemented a broad interface with a large number of capabilities. HORUS reorganized the interface and eliminated several of the more complex functions. ENSEMBLE went even further in this direction. Spread, in contrast, has an extremely simple interface and a limited set of basic capabilities. After the tradeoffs were considered the following characteristics were chosen for the AL API:

b. The AL API will support information hiding such that no underlying data structures associated with the underlying process group communications package would be used at the interface. In addition the interface will be designed such that any of the AL's underlying structures can be changed without impacting the AL API or any applications using it.

(1) The AL API will be multithreaded and will use up-call mechanisms to deliver information to the application.

(2) The AL API will support messages constructed of an unlimited number of variable sized buffers.

(3) The AL API will support the following types of ordering:

(a) FIFO ordering.

[3] The "one endpoint per application" rule is artificial in that Spread can support multiple connections from a single application. The AL does not support multiple connections from a single application. There can only be one AL library linked in an application, and the AL only supports a single connection call.

(b) Causal ordering – Consider the situation where message 1 is delivered to processes A and B and A responds by sending message 2 on to process B. Causal ordering guarantees that B will receive message 1 followed by message 2. This is illustrated in Figure E-10.

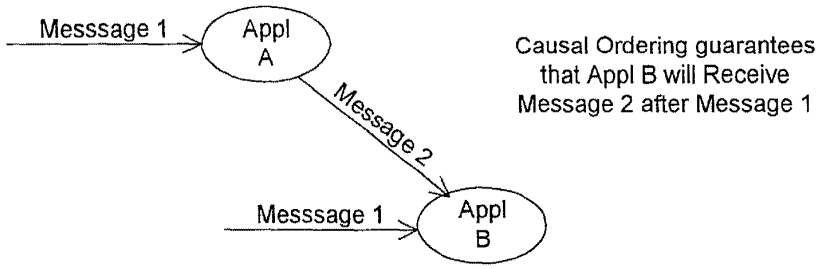


Figure E-10 Causal Ordering

(c) Total ordering – Assures that all receivers receive all messages in the exact same order. This is more strict than causal ordering in that causal ordering deals with message exchanges that have a common node, and total ordering applies to all communications, even communications over parallel but unrelated paths. This is illustrated in Figure E-11.

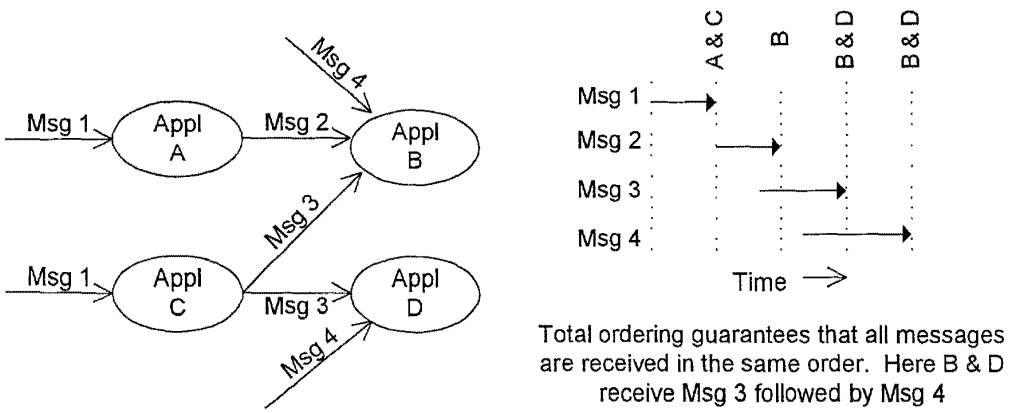


Figure E-11 Total Ordering

(4) The AL API will support the following primary functions:

(a) Group join and leave

(b) Multicast send

(c) Unicast send

(d) Multicast RPC – RPC Send calls require the receiver to reply explicitly to the message. The AL manages the replies and notifies the sender when all replies have been received. It also manages situations where expected replies can never be received because a message receiver leaves the group before replying.

(e) Unicast RPC

(f) Multicast stability – The sender is notified when the AL determines that all receivers have received the message.

(g) Unicast stability

(h) Message and buffer management routines.

(5) The API will support the following required up-calls:

(a) Message reception

(b) Endian conversion

(c) View change

(6) The API will support the following optional up-calls:

(a) RPC reply received

(b) Stability notification

c. State transfer mechanisms are explicitly not included in the definition of the AL API because none of the candidate process group communications packages supported synchronized state transfer. If state transfer were written into the AL, it would have to be redeveloped when a new middleware package is selected. To avoid this the state transfer capability was written into the modified GMS.

d. An initialization routine is provided in the AL API. In the Spread AL implementation, this is where the connection is made to the Spread daemon. The AL API requires that this call be made before any other call in the API.

e. In using the AL API, a user joins a group by making a call to a join routine. In this call, the user specifies functions that are to be called when a view change occurs, when an endian conversion is required, and when a data message is received. No further action needs to be taken to receive a message. The next message received in the group will result in the data received function up-call being made.

f. To send a message, the application using the AL must first make an explicit call to create the message. In this context, a message consists of a header that contains information about the message and data space that contains the data that the user sends with the message. The create message function creates the header and returns a pointer to that header. It does not allocate storage space for the data associated with the message. The application then creates buffers and appends them to the message. This is done by passing the address and size of a piece of application-owned storage to create a buffer function. This approach prevents forcing the application to copy data into a contiguous space before sending the message. The AL passes the buffer pointer and size information down to Spread. Spread then uses this information to form the message packet that is transmitted over the network.

E.6 AL Design

An adaptation layer has been implemented to connect the AL API to the Spread process group communications package. One of the major issues that had to be addressed in this development was that Spread implements a standard down-call mechanism for receiving messages. The AL API defines an up-call mechanism for receiving information. This conflict was addressed by using a multithreaded design for the AL.

E.6.1 Threading Model

a. The thread design inside the AL is driven by the difference in calling paradigms between Spread and the AL API. A single *Spread listen* thread is established that blocks on a Spread receive call and waits for messages to be delivered. An additional thread is created each time the application joins a group. A new message queue is also established for each group joined. The group thread monitors this queue for new messages. When the *Spread listen* thread receives a message from Spread, it places the message on the new message queue that is associated with the group in which the message was received. The threading structure is illustrated in Figure E-12.

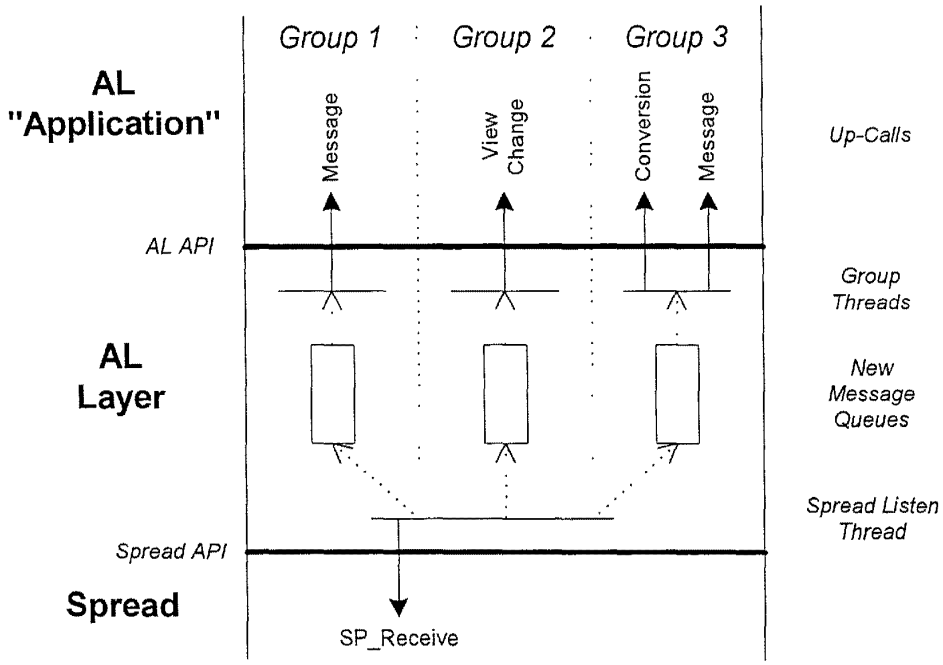


Figure E-12 AL Thread Model

b. When the group thread gets a message on its new message queue, it first identifies the message type. If the message is a view change, the group thread calls the view change function defined by the application in the join call. If the message is a data message, the group thread first checks to see if the sending machine is of a different endian architecture than the receiving machine. If this is the case, the message is passed up to the application as an argument to the conversion up-call. This gives the application an opportunity to correct for endian differences before the message is processed. The data message up-call is made after any conversion up-calls are complete.

c. The AL thread model allows the application to receive messages from multiple groups asynchronously. Within a single group, however, only one up-call will be made to the application at any time. The AL layer maintains message order. Up-calls will be made in the order that the messages generating the up-call arrived. There is no ordering supported between groups. This means that a message destined for group A that arrives after a message destined for group B could be delivered to group A before the other message is delivered to group B.

E.6.2 Data Structures

The data structures in the AL consist primarily of doubly and singly linked lists. These structures are described in the subsections that follow. The relationships among the data structures are illustrated in Figure E-13.

E.6.2.1 Group List

The primary data structure is the group list. This is a doubly linked list that contains an entry for every group joined by the application. Each entry contains pointers to the new message queue for that group, a pointer to the current view for the group, and pointers to a list of records that identify any replies expected from other members of the group. Records in this list also contain all configuration information associated with the group including the default message ordering to be used, the group name, and the up-call functions defined for the group.

E.6.2.2 New Message Queue

a. The new message queue is used as common storage between the *Spread listen* thread and the group thread. The group thread reads records from the top of the queue and the *Spread listen* thread places new messages on the bottom of the queue. There is one of these structures for every group joined by the application. The head and tail pointers for this structure are held in the group record (component of the group list) of the group associated with the queue.

b. If the new message queue entry represents a view change, a pointer to an array is established when *Spread listen* creates the queue entry. Each entry in this array is a pointer to an endpoint record for a member of the new view of the group. If the new message queue entry represents a data message, a pointer to the message structure for the message is established.

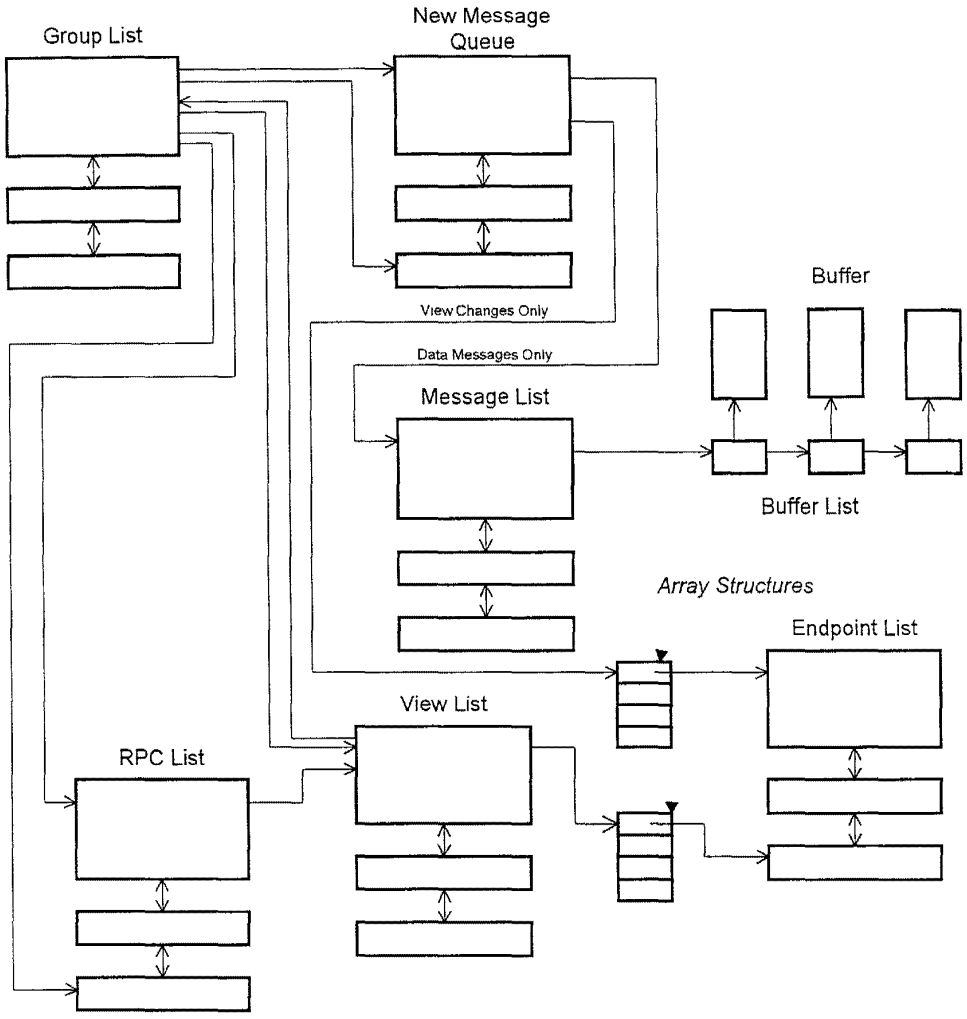


Figure E-13 AL Data Structures

E.6.2.3 Message List

The message list structure is a doubly linked list that contains all of the messages that exist at any single point in time in the system. Messages are linked in a list to provide easy access by diagnostic routines that are capable of printing all messages in the system. Messages are not related by their position in the linked list. Message records are accessed either through direct reference via message handles held by the application or through access via the new message queue associated with a particular group.

E.6.2.4 Buffer List

A message list record contains all of the information that pertains to a specific message, including a pointer to a list of buffers that hold the data associated with the message. If the message was created by the application in preparation for transmission, there will be an entry for each buffer appended to the message by the application. If the message was created by *Spread listen* when a message was received from Spread, then there will be only one buffer and that will contain all of the data associated with the message. Each buffer record in the list pointed to by a message list record contains the memory address of the actual data storage (in effect, this is a pointer to the storage), the size of the data storage, and a pointer to the next buffer record.

E.6.2.5 View List

A record in the group list contains a pointer to a record in the view list that represents the current view for that group. The view list is a doubly linked list of all views (of all groups) that are currently active. This list also holds any expired views that are still being referenced by the application (see Section **Error! Reference source not found.**, which describes reference counts). Each view record contains all of the information that relates to the view including a pointer to an array. Each entry in this array contains a pointer to the endpoint record of a group member that is present in the view.

E.6.2.6 RPC List

Records in the group list also contain head and tail pointers to an RPC list. There is an entry in the RPC list for each outstanding RPC message or stability message that has been sent in the group. Each entry contains the up-call routines that are to be called when an RPC reply is received or a stability response is determined. Each record also contains a pointer to the view in which the RPC or stability request was made. This is used to determine the endpoints from which replies are expected.

E.6.3 Reference Counts

a. The user of the AL API receives and manages handles for objects created and managed by the AL. These handles are implemented as pointers to isolate the AL API and its users from changes in the AL layer. Because the AL creates the objects pointed to by these handles, it must also be responsible for removing them. It is expected that the AL users will need to create copies of these handles. This creates a conflict because the AL layer needs to know when copies have been made so that it can know when a structure is no longer needed and can be removed.

b. To address this problem, the concept of reference counted objects was created. When the AL user copies a reference counted object, the user must call a routine to increment a reference counter associated with that object. The AL also increments this count when the object is in use by the AL. When the AL is finished with the object, it decrements the reference count. When the AL user is finished with the copy of the object, the user must decrement the reference count. When the reference count goes to zero, the object is removed. The reference counted objects in the AL API are endpoints, groups, messages, and views.

E.6.4 Transmission Formats

a. When the AL makes a send call to Spread, the AL passes Spread the name of the group in which the message will be sent and a data message that is encoded according to the type of message that is being sent. The encoding used is shown in Figure E-14.

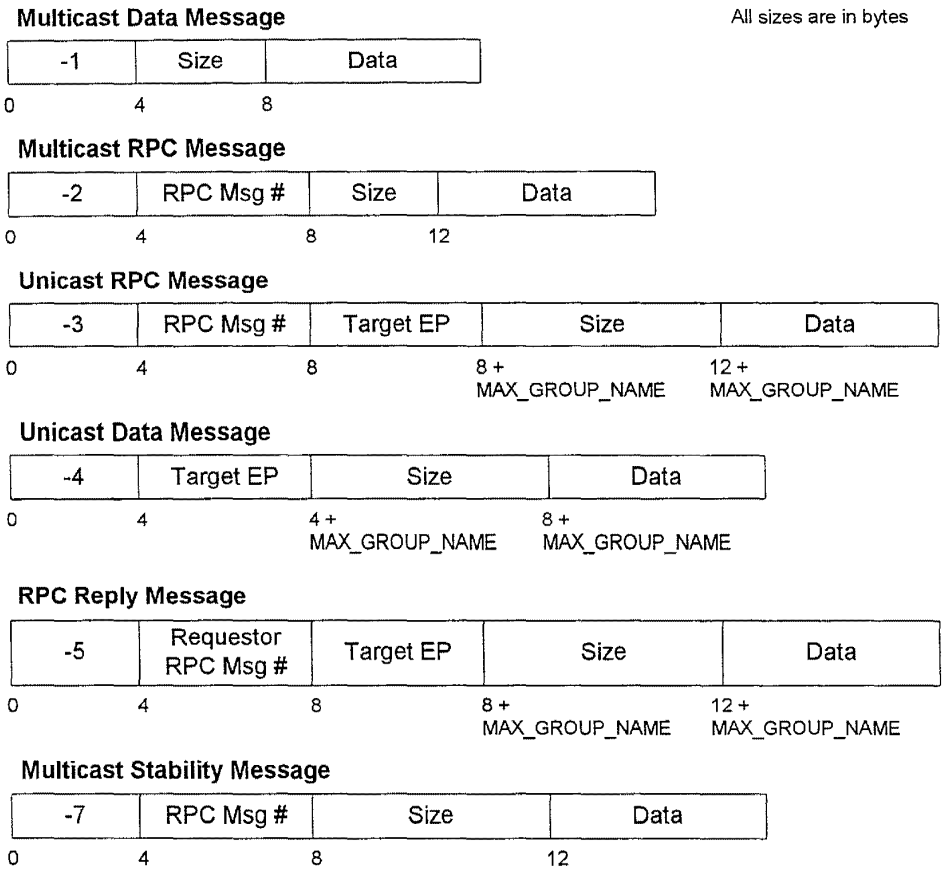


Figure E-14 Data Formats

b. The first 4 bytes in each format contain a negative integer that identifies the message type. In all formats the data area is preceded by a 4-byte integer that contains the size of the data area in bytes. The *Target EP* field contains the name of the unicast message destination endpoint. Receivers ignore the message if the *Target EP* field does not contain its name. The *RPC Msg #* field in the RPC messages and in the Multicast Stability Message contain an integer value that, in combination with the identity of the requesting process, uniquely identifies the RPC request. This value is echoed back in the *Requestor RPC Msg #* field in the RPC Reply Message.

c. The data areas in each message format contain the buffers that are associated with the message. This is illustrated in Figure E-15.

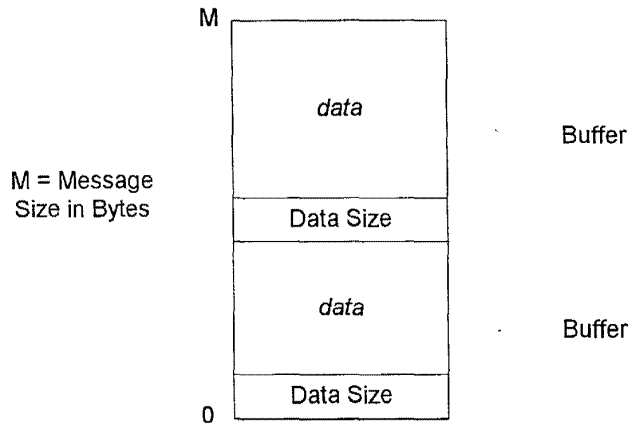


Figure E-15 Transmitted Data Message

E.7 Transformers

a. The magnitude of the AL development effort was such that it was a high-risk item for the 1998 demonstration. It was not clear that the effort could be completed, integrated with HiPer-D, and integrated at NSWCDD in time for the 1998 demonstration. To reduce the risk, the decision was made to focus only on the integration of JHU/APL's components for the 1998 demonstration. NSWCDD components would continue to use ISIS. Transformers would be used to bridge between process groups implemented in Spread and their colleagues implemented in ISIS.

b. A transformer is a standalone process that exchanges messages between a Spread group and an ISIS group. Transformers are built with both communications stacks and simply exchange messages at the top of each stack. Transformers do not implement any of the state transfer protocols. They are strictly limited to basic message exchange.

c. With one exception, a separate transformer was built for each group that was used by both NSWCDD and JHU/APL components. The exception was the RTDS. The structure of the RTDS is such that it was easier to convert its inputs to use the new AL directly. The new RTDS gets information from the ATCF through the AL layer and distributes the information to its clients using ISIS.

E.8 Results

a. Development and initial integration were completed at JHU/APL. A version of JHU/APL's components was delivered to the lab at NSWCDD and integrated in time for the 1998 HiPer-D demonstration. While several problems were uncovered and corrected during integration at JHU/APL, no problems were encountered in the AL, DCS, or GMS during the integration efforts at NSWCDD. Two problems were detected and corrected in the transformers during this integration. No quantitative comparisons were made, but performance of the new system appears to be at least comparable to the ISIS based system.

b. Considering the fact that this effort significantly changed the underlying middleware of all of JHU/APL's HiPer-D components, this effort can only be considered a tremendous success. The accomplishment is even more striking when it is remembered that the effort started relatively late in the year and that developments in other HiPer-D components were carried out in parallel.

Adaptive QoS and Resource Management Using *A Posteriori* Workload Characterizations[1]

Lonnie R. Welch[†], Paul V. Werme[‡], Binoy Ravindran[∫], Larry A. Fontenot[‡], Michael W. Masters[‡], D. Wayne Mills and Behrooz A. Shirazi[†]

[†]*Computer Science and Engineering Dept.*
The University of Texas at Arlington
Arlington, TX 76019-0015
{welch|shirazi}@cse.uta.edu

[‡]*The Naval Surface Warfare Center*
Dahlgren, VA 22448
{WermePV|MastersMW|FontenotLA|Mills DW }@nswc.navy.mil

[∫]*The Bradley Department of Electrical and Computer Engineering*
Virginia Polytechnic Institute and State University
binoy@ee.vt.edu

Abstract

Certain real-time applications must operate in highly dynamic environments (e.g., battle environments), thereby precluding accurate characterization of the applications' workloads by static models. Thus, guarantees of real-time performance based on a priori characterizations are not possible. However, potential benefits of a posteriori approaches are significant, including the ability to function correctly in dynamic environments (through adaptability to unforeseen conditions), and higher actual utilization of computing resources.

In this paper, we present an approach that is appropriate for systems which experience large variations in workload. A distributed collection of computing resources is managed by continuously computing and assessing QoS and resource utilization metrics that are determined a posteriori. The utility of our approach is shown by applying it to a large, experimental distributed Navy computing system.

1 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 [13]. 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*. 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*, which are described in [6] as follows:

> *"Modern naval combatants host many highly complex systems. Each system performs one or more tactical capabilities. The single large-scale system formed via integration of these complex systems is a Combat System. ...*
>
> *The combat system processing demand per unit of time is defined as follows. Each tactical capability, e.g., track management, has its own processing*

[1] 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.

demand. This demand is dependent on the number of objects, e.g., tracks, that will utilize this capability. The total number of capabilities active concurrently varies with time. The total number of objects driving each capability varies with time. Thus, the combat system processing demand per time unit is dependent on number of objects per capability in the time unit and the number of capabilities active during the time unit. ...

A definition of the demand space features and supply space features is needed. The mapping of demand space features onto the supply space features is needed. Indices that support clustering or partitioning of demand features on or across supply features also is needed. ..."

Implications of these requirements are that demand space workload characterizations may need to be determined *a posteriori,* and an adaptive approach to resource allocation may be necessary. 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 [10, 12, 21, 19, 18, 13, 3]. While [13] establishes the utility of WCET-based approaches by listing their domains of successful application, others [9, 7, 5, 8, 16, 12, 17, 15, 14, 11, 1, 2, 4] cite the drawbacks, and in some cases the inapplicability, of the approaches in certain domains. In [12, 17, 9, 5, 1] 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 [14, 1] 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 [7, 15, 14]. Paradigms which generalize the execution time model have also been developed. Execution time is modeled as a *set* of discrete values in [8], as an interval in [16], and as a probability distribution in [9, 17, 2]. Most models consider execution time to apply to the job atomically; however, some paradigms [11, 15] view jobs as consisting of mandatory and optional portions; the mandatory portion has an *a priori* known execution time in [11], and the optional portion has an *a priori* known execution time in [15]. 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 [5] *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. The approach most similar to the one presented in this paper is described in [4], where resource requirements are observed *a posteriori,* allowing applications which have not been characterized *a priori* to be accommodated. Also, for those applications with *a priori* characterizations, the observations are used to refine the *a priori* estimates. These characterizations are then used to drive resource availability based algorithmic and period variation within the applications.

In this paper we present an approach that is appropriate for systems which experience large variations in workload. We elaborate the details of the language, system model, metrics, and middleware presented in [20]. Furthermore, the experimental results presented in [20] were performed on a benchmark system, whereas this paper presents results from applying the technology within an experimental Navy distributed computing system. The techniques are used during war-fighting test scenarios, demonstrating processing of up to 7500 radar tracks while meeting real-time requirements. The testbed consists of many application systems (greater than 100 processes consisting of more than 1 million lines of source code) being managed by the middleware, under highly dynamic system workloads, and on many cooperating hosts (40 hosts with 60 processors). These experiments provide a proof of concept for the specification language and the a posteriori techniques for modeling, monitoring and resource allocation. The results also demonstrate very fast (sub-second) detection and reallocation services for large-scale systems.

2 The System Model

Our approach to adaptive resource and QoS management is based on the dynamic path paradigm. A path-based real-time subsystem (see [20]) 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. A system operates in an environment that is either deterministic, stochastic, or dynamic. A deterministic environment exhibits behavior that can be characterized by a constant value. A stochastic environment behaves in a manner that can be characterized by a statistical distribution. A dynamic environment (such as a war-fighting environment) depends on conditions which cannot be known in advance.

For example, an air defense subsystem can be modeled using three dynamic paths: *threat detection, engagement,* and *missile guidance.* The *threat detection* path examines radar sensor data (radar tracks) and detects potential threats. The path consists of a radar sensor, a sensor data stream, a filtering program and an evaluation program. When a threat is detected and confirmed, the

engagement path is activated, resulting in the firing of a missile to engage the threat. After a missile is in flight, the *missile guidance* path uses sensor data to track the threat, and issues guidance commands to the missile. The *missile guidance* path involves sensor hardware, software for filtering, software for evaluating & deciding, software for acting, and actuator hardware.

The approach described in this paper pertains to detection & assessment paths. This type of path *continuously* evaluates the elements of a sensor data stream to determine if environmental conditions are such that an action should be taken. Thus, this type of path is called *continuous*. Typically, there is a timeliness objective associated with completion of one review cycle of a continuous path, i.e., on the time to review all of the elements of one instance of a data stream. (The data stream is produced by sampling the environment. One set of samples is the data stream instance.)

The *threat detection* path of an air defense system is an example of a continuous path. It is a sensor-data-stream-driven path, with desired end-to-end cycle latencies for evaluation of radar track data. If it fails to meet the desired timeliness quality of service in a particular cycle, the path must continue to process track data, even though desired end-to-end latencies cannot be achieved. Peak loads cannot be known in advance for the *threat detection* path, since the maximum number of radar tracks that may exist in a battle environment cannot be known *a priori*. Furthermore, average loading of the path is not a useful metric, since the variability in the sensor data stream size is very large - it may consist of zero, 10s, 100s or 1000s of tracks.

We have developed a demand space model based on the dynamic real-time path paradigm. A software subsystem, **SS**, consists of (1) a set of applications ($\mathbf{SS.A} = \{ a_1, a_2, \ldots \}$), (2) a set of devices (sensors and actuators) ($\mathbf{SS.D} = \{ d_1, d_2, \ldots \}$), (3) a communication graph defining the connectivity between applications and devices ($\Gamma(\mathbf{SS}) \in \Pi((\mathbf{SS.D} \cup \mathbf{SS.A}) \times (\mathbf{SS.D} \cup \mathbf{SS.A}))$), and (4) a set of paths ($\mathbf{SS.P} = \{P_1, P_2, P_3, \ldots\}$). (Note: Π denotes the power set).

Each *continuous path* $\mathbf{P_i}$ is represented as (1) a set of applications $\mathbf{P_i.A} = \{a_{i,1}, a_{i,2}, \ldots\}$ (where $\mathbf{P_i.A} \subseteq \mathbf{SS(P_i).A}$), (2) a set of devices $\mathbf{P_i.D} = \{d_{i,1}, d_{i,2}, \ldots\}$ (where $\mathbf{P_i.D} \subseteq \mathbf{SS(P_i).D}$), (3) a communication graph $\gamma(\mathbf{P_i}) \in \Pi((\mathbf{P_i.D} \cup \mathbf{P_i.A}) \times (\mathbf{P_i.D} \cup \mathbf{P_i.A}))$ (note that $\gamma(\mathbf{P_i}) \subseteq \Gamma(\mathbf{SS(P_i)})$), and (4) a data stream $\mathbf{P_i.DS}$. (Note: $\mathbf{SS(P_i)}$ denotes the subsystem in which path $\mathbf{P_i}$ is contained.) $\mathbf{Profile(a_i)}$ is the set of hosts where application 'a_i' is eligible to be run (i.e., the set of hosts for which a_i has been compiled). For the communication graph $\gamma(\mathbf{P_i})$, the head node of the graph (which is the application which receives the initial input data stream) is represented as $\mathbf{ROOT(P_i)}$, and the last node of the graph (which is the application which communicates with other applications or paths outside of $\mathbf{P_i}$) is represented as $\mathbf{SINK(P_i)}$. The type of $\mathbf{P_i}$'s data stream is defined as $\tau(\mathbf{P_i.DS}) \in$ {dynamic, stochastic, deterministic}. (For the remainder of this paper, it is assumed that the all data stream types are dynamic).

The real-time QoS requirements of a continuous path include one or more of the following: (1) required latency of $\lambda_{\mathbf{REQ}}(\mathbf{P_i})$ seconds, (2) required throughput of $\theta_{\mathbf{REQ}}(\mathbf{P_i})$ data stream elements per second, and (3) required data inter-processing time of $\delta_{\mathbf{REQ}}(\mathbf{P_i})$ seconds (the maximum allowable time between processing of a particular element of $\mathbf{P_i.DS}$ in successive cycles). To mask transient QoS violations during QoS monitoring, a specification may also define a sampling window and a maximum number of QoS violations to be tolerated within the window; $\omega(\mathbf{P_i})$ models the sampling window size and $\upsilon(\mathbf{P_i})$ represents the maximum allowable number of violations within the sampling window.

The demand space model also captures information that must be obtained *a posteriori*. Some application programs can be replicated for load sharing. The set of replicas of application '$a_{i,j}$' during cycle 'c' of $\mathbf{P_i}$ is defined as $\mathbf{REPLICAS}\ (\mathbf{a_{i,j}, c}) = \{\mathbf{a_{i,j,1}}, \mathbf{a_{i,j,2}}, \ldots\}$. The host to which application '$a_{i,j,k}$' is assigned during cycle 'c' of path $\mathbf{P_i}$ is defined as $\mathbf{HOST}\ (\mathbf{a_{i,j,k}, c, P_i})$.

The set of elements that constitutes a data stream can vary dynamically. $\mathbf{P_i.DS(c)} = \{\mathbf{P_i.DS(c)}_1, \mathbf{P_i.DS(c)}_2, \ldots\}$ represents the set of elements in $\mathbf{P_i.DS}$ during cycle 'c' of $\mathbf{P_i}$. The **tactical load** (in number of data stream elements processed) of a continuous path $\mathbf{P_i}$ during it's c^{th} cycle is $|\mathbf{P_i.DS(c)}|$. The processing of elements of a data stream may be divided among replicas of an application to exploit concurrency as a means of decreasing execution latency of a path. In successive stages of a path that has non-combining applications (applications which, after processing data received from a single predecessor, simply divide the data among their successors), data will arrive in batches to applications; hence, each application may process several batches of data during a single cycle. Thus, the model represents the set of elements from *all* batches of data processed by application/replica 'a' during cycle 'c' as $P_i.DS(c, a) = \{P_i.DS(c, a)_1, P_i.DS(c, a)_2, \ldots\}$ The cardinality $|P_i.DS(c, a)|$ is the tactical load of 'a' in cycle 'c'. The data stream elements contained in the jth batch of 'a' are denoted by $P_i.DS(c, a, j) = \{P_i.DS(c, a, j)_1, P_i.DS(c, a, j)_2, \ldots\}$.

3 QoS Specification

This section presents a specification language for describing the characteristics and requirements of dynamic, path-based real-time systems, and incorporates the system model constructs described in Section 2. The language provides abstractions to describe the properties

of the software, such as hierarchical structure, interconnectivity relationships, and run-time execution constraints. It also allows description of the physical structure or composition of the hardware such as LANs, hosts, interconnecting devices (such as bridges, hubs, and routers), and their statically known properties (e.g., peak capacities). Further, the quality of service requirements of various system components can be described. The constructs of the specification language are illustrated within the context of the distributed experimental Navy combat system components (described in Section 5).

A high-level system specification is shown in Figure 1. At the highest level, a specification consists of a collection of software systems, hardware systems, and network systems. A software specification is a collection of software systems, each of which consists of zero or more software subsystems (**SS**). This is illustrated in Figure 1 for the AAW system. The Doctrine SS has a priority, a set of dynamic real-time path definitions (**SS.P**), and a set of application program definitions (**SS.A**).

A path ($\mathbf{P_i}$) is defined as a *connectivity* graph ($\gamma(\mathbf{P_i})$) of constituent applications, a set of attributes (priority and type), QoS requirements, and data/event stream definitions ($\mathbf{P_i.DS}$). The connectivity specification represents the communication relationships among applications ($\mathbf{P_i.A}$) in a path. These relationships form a directed graph, specified as a set of ordered application pairs, which indicates the primary data flow between applications. (Note that the names of the applications specified in the sample path are fully qualified as system:subsystem:application.)

A path's real-time QoS requirement specification may include simple deadlines, inter-processing times, throughputs, and super-period deadlines. A simple deadline is defined as the maximum end-to-end path latency ($\lambda_{REQ}(\mathbf{P_i})$) during a cycle of a continuous or quasi-continuous path, or during an activation of a transient. Inter-processing time ($\delta_{REQ}(\mathbf{P_i})$) is defined as a maximum allowable time between processing of a particular element of a continuous or quasi-continuous path's data stream in successive cycles. The throughput requirement ($\theta_{REQ}(\mathbf{P_i})$) is defined as the minimum number of data items that the path must process during a unit period of time. Each timing constraint specification may also include items that relate to the dynamic monitoring of the constraint (e.g., slack).

The scalability specification indicates whether the path is scalable, and includes specifications for defining when and how scaling will be accomplished. (For a path to be scalable, one or more applications in the path must be scalable.) A datastream specification ($\mathbf{P_i.DS}$) is also given for the sample path. The stream type ($\tau(\mathbf{P_i.DS})$) can be deterministic, stochastic, or dynamic.

```
HARDWARE SYSTEM Navy_Ship {...}
NETWORK SYSTEM Ship_Net {...}
SOFTWARE SYSTEM ATWCS {...}
SOFTWARE SYSTEM AAW
{ // This line is a comment
  SUBSYSTEM Displays {...}
  SUBSYSTEM TacticalServices {...}
  SUBSYSTEM Doctrine {
    Priority 2;
    PATH AutoSpecial_Review {
      Connectivity {
        (AAW:TrackServices:Track_Processor, AAW:TrackServices:RTDS);
        (AAW:TrackServices:RTDS, AAW:Doctrine:AutoSpecial);
        (AAW:Doctrine:AutoSpecial, AAW:TacticalServices:Engage_Server);
        (AAW:TacticalServices:Engage_Server, AAW:TacticalServices:WCS);
      } //end Connectivity
      Type Continuous;
      Priority 1;
      RealTimeQoS {
            SimpleDeadline          65.0;
            InterProcessingTime     0.600;
            Throughput              200;
            BatchLatency            20.0;
            BatchInterArrival       550.0;
            MaxSlack                80;
            MinSlack                20;
            SlidingWindowSize       20;
            Violations              15;
      } //end of real-time QoS definition
      Scalability { Scalable TRUE;
                    PathSettlingTime 30.00;}
      DATASTREAM { Type Dynamic;
                   SlackQoS 400;}
    } //end path AutoSpecial_Review
    PATH SemiAuto_Review {...}
    Application AutoSpecial {...}
    Application SemiAuto {...}
  } //end of software subsystem Doctrine
} //end of software system AAW
```

Figure 1. AAW sys & AutoSpecial_Review path.

The applications ($\mathbf{P_i.A}$) which constitute a path are included in the QoS specifications (Figure 1 & 2). An application is either an executable image that may be started as an autonomous process on a host, or a script file that potentially forks multiple processes. An application has several control characteristics. First, there are time delays associated with the application (representing the amount of time the control program must wait before starting this application, and the time the application requires to complete its initialization once it has been started). There are also definitions controlling how an application is started. The *Automatic* attribute is used to determine which applications should be started automatically as part of default system initialization. The *RMStart* field specifies whether the Resource Management infrastructure should decide where the application should be started or whether a static configuration should be used. *Console* and *Display* are attributes pertaining to the graphical capabilities required by the application (mainly used for debugging). *Memory* indicates the minimum amount of memory that the application requires in order to execute.

An application can have one or more *Startup* blocks to describe the resource requirements of the application. This information includes required host type and operating system type and version(s); alternately, this may be an optional list of hostnames. The startup information also includes the working directory, name of the executable, and an *ordered* list of arguments. (Multiple *Startup* blocks are used when multiple versions of an application exist, such as executables compiled for different machine architectures.) The *Shutdown* block specifies either a signal used to shutdown the application, or a script to gracefully terminate the application.

The *Dependency* block describes temporal relationships between applications, including startup and shutdown dependencies and time delay requirements.

A survivability QoS specification includes a Boolean variable that indicates (1) whether the application should be managed to ensure survivability (i.e., fault tolerance) and (2) the minimum required level of redundancy. The flag, *SameHost*, indicates whether the application must be started on the same host on which it failed (which is useful for providing fault tolerance for daemons that must run on a specific host).

The *scalability* specification indicates if an application can be scaled via replication. (Currently, applications specified as scalable are assumed to be capable of performing load sharing among replicas, and adapting automatically to varying numbers of replicas.)

Hardware system specifications (Figure 3) allow the description of zero or more hardware subsystems. Each hardware subsystem consists of one or more hosts. A host specification describes the host's name, type, operating system and version, number of processors, processor speeds, RAM capacity, and network connectivity. Network system specifications describe the networks and interconnection devices such as switches, hubs, and routers. A network system consists of zero or more subsystems which may contain networks (each with an associated peak bandwidth specification) and/or interconnection devices (each containing a description of network membership). (Hardware and network system models employed for characterizing the resource supply space features are described in Section 4.)

```
APPLICATION AutoSpecial {
  TimeDelay 4; // Wait this long (secs) before starting
  SettlingTime 5; // App needs this long to initialize to steady state.
  Automatic TRUE; // Automatically start this app
  RMStart TRUE; // This application is to be allocated a host by RM
  Console TRUE;        // Start this App in a XTerm
  Display "tactical1";   // Where to place the display for this App
  Memory 2; // Min amount of Free RAM (Mb) needed
  STARTUP {
    Type   "SUN"; OS "Solaris";
    Version "2.5.1"; Version "2.6";
    Directory "$TACTICAL_BINDIR";
    Execute "auto_special.solaris.exe";
    Arg "%(UNIQUE, 1, 32)";
    Arg "> /usr/tmp/%(USERID)_auto_spec_%(UNIQUE, 1, 32).out";
  } //end STARTUP
  SHUTDOWN { Script "auto_special_shutdown"; }
  DEPENDENCY {
    Type STARTUP; // STARTUP, SHUTDOWN
      Name "AAW:TacticalServices:Doctrine_Server";
      Delay 10; // seconds
  }
  RestartDelay 1; // Time to delay starting after a detected failure
  SurvivabilityQoS {    Survivable TRUE;
                        MinCopies 1;
                        SameHost FALSE;
  }
  Scalability {         Scalable TRUE;
                        Combining FALSE;
                        Splitting FALSE;
  }
}
```

Figure 2. Specification of AutoSpecial.

4 Adaptive QoS and Resource Management

This section defines metrics and techniques for reasoning about the mapping of demand space onto supply space, i.e., for resource and QoS management. Our approach (depicted in Figure 4) works as follows. Application programs of real-time control paths send time-stamped events, via the *Application Instrumentation* component, to the *Path QoS Monitor* component. The *Path QoS Monitor* component calculates path- and application-level QoS metrics, compares observed QoS to required QoS, and notifies the *QoS Diagnosis* component when QoS violations are detected. The *Host & Network Monitoring* component collects operating system and network performance, status, and load information, which is then provided to the *Resource QoS Monitor* component. Here, host and network statistics are correlated, performance and load histories are maintained, and load metrics are calculated. This information is made available to the *QoS Diagnosis* component for use in determining resource loading, and allocation tradeoffs. The *QoS Diagnosis* component determines the cause of QoS violations, analyzes and ranks potential reallocation actions for restoring required QoS, and provides this list of recommended actions along with associated

```
HARDWARE SYSTEM Navy_Ship {
  SUBSYSTEM Compartment_5 {
    HOST tactical_12 {
      Type "SUN";
      OS "Solaris"; Version "2.6";
      Speed 250; //MHz Memory 256; //MB
      NumCPUs 2;
      Default-Network Ship_Net:Ether_100;
      Network Ship_Net:ATM_250;
    }
    HOST tactical_13 {...}
  }
  SUBSYSTEM Compartment_6 {...}
  SUBSYSTEM Compartment_7 {...}
}
NETWORK SYSTEM Ship_Net {
  LAN Ether_100 { Bandwidth 100;}
  LAN ATM_250 {Bandwidth 250;}
  IC Hub_3 {
    Network Ship_Net:Ether_100;
    Network Ship_Net:ATM_250;
  }
}
```

Figure 3. Host & net. specs.

host and network load metrics to the *Resource Allocation* component. The *Resource Allocation* component determines the most beneficial allocation of resources for restoring required QoS. The allocation actions selected are then implemented by the *Application & Resource Control* components. These techniques are explained in more detail in this section.

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

Observed real-time QoS metrics are defined in terms of these basic events as follows: (1) latency of path P_i during cycle 'c' is $= \lambda_{OBS}(P_i,c) = \mathbf{max}(\{e(P_i.DS(c, a_{i,m,n}, j)) - s(P_i.DS(c, a_{i,x,1}, 1)) \mid a_{i,m} = SINK(P_i), a_{i,x} = ROOT(P_i)\})$ (note that λ_{OBS} is the maximum value from the *set* of latencies of all batches of data processed by all replicas of $SINK(P_i)$ during the cycle), (2) data-inter-processing time of application 'a' in path P_i during cycle 'c' of data stream $P_i.DS(c,a)$ is approximated as $\delta_{OBS}(P_i.DS(c,a)) = \{s(P_i.DS(c,a)) - s(P_i.DS(c-1,a))\}$, for c > 1, (3) data-inter-processing time of path P_i during cycle 'c' for data stream $P_i.DS(c,a)$ is $\delta_{OBS}(P_i.DS(c)) = \delta_{OBS}(P_i.DS(c,a))$, where 'a' = $ROOT(P_i)$, (4) observed cycle throughput of path P_i during cycle 'c' is $\theta_{OBS}(P_i,c) = |P_i.DS(c)| / \lambda_{OBS}(P_i,c)$, (5) workload of application/replica 'a' of path P_i during cycle 'c' is $W_{OBS}(P_i,c,a) = |P_i.DS(c,a)| / \delta_{OBS}(P_i.DS(c,a))$, and (6) workload of path P_i during cycle 'c' is $W_{OBS}(P_i,c) = |P_i.DS(c)| / \delta_{OBS}(P_i.DS(c)) = (\Sigma|P_i.DS(c,a_{i1k})|) / \delta_{OBS}(P_i.DS(c))$, for all replicas k of $ROOT(P_i)$.

Analysis of a time series of the real-time QoS metrics enables detection of QoS violations. An **overload** of a path or application occurs in any cycle 'c' where the number of violations within the sample window $\omega(P_i)$ equals or exceeds the maximum number of violations $\upsilon(P_i)$. As an example, detection of a path-level QoS latency violation occurs when the observed path latency $\lambda_{OBS}(P_i)$ exceeds the required path latency $\lambda_{REQ}(P_i)$ for $\upsilon(P_i)$ samples within the sample window of the most recent $\omega(P_i)$ samples. This can be expressed as $\upsilon(P_i) \leq |\{d: (c-d)+1 < \omega(P_i) \wedge [(\lambda_{REQ}(P_i) - \lambda_{OBS}(P_i,d)) < 0)]\}|$, where 'c' is the current data stream cycle and 'd' represents data stream cycles within the sliding window $[c-(\omega(P_i)-1), c]$. For the experiments described in Section 6, path latencies ($\lambda_{OBS}(P_i, c)$) are used for determining QoS violations.

The components of the supply space are also modeled *a posteriori* in our approach. A hardware system, **HS**, consists of (1) a set of hosts $HS.H = \{h_1, h_2, \ldots\ldots\}$, (2) a set of Local Area Networks or LANs, $HS.L = \{L_1,L_2, \ldots\ldots\}$, and (3) a set of interconnecting devices $HS.I = \{i_1,i_2, \ldots\ldots\}$. The system model captures several hardware load metrics. The *paging score* of a host h_i at time t is defined as $PS(h_i,t)$, and is calculated as the number of page faults per second averaged over the time interval t_1, divided by a maximum page fault threshold. The *cpu score* of a host h_i at time t is defined as $CS(h_i,t)$, and represents

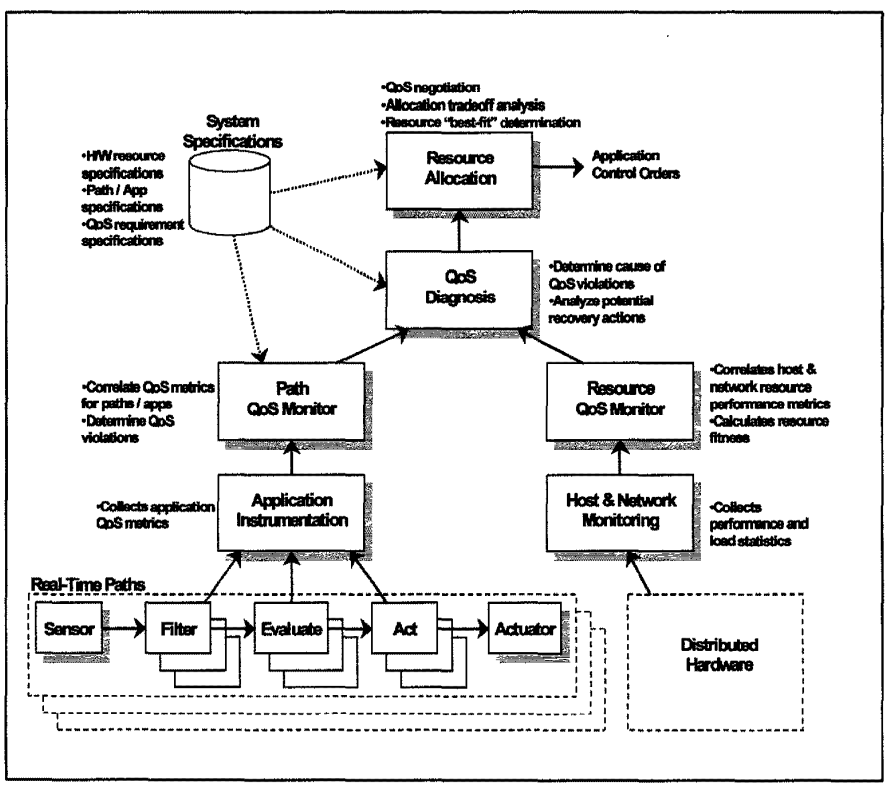


Figure 4. QoS & resource management.

the average percent CPU idle time over time interval t_2. The *network score* of a host h_i at time t is defined as $\mathbf{NS(h_i,t)}$, and is calculated as the number of packets received plus the number of packets sent averaged over time interval t_3, divided by a maximum network packet threshold. (All scores fall within the interval [0,1].)

Fitness scores for each of the host load metrics are calculated as follows: The *paging fitness* is calculated as $\mathbf{PF(h_i,t) = (1 - PS(h_i,t))}$. The *cpu fitness* is calculated as $\mathbf{CF(h_i,t) = CS(h_i,t)}$. The *network fitness* is calculated as $\mathbf{NF(h_i,t) = (1 - NS(h_i,t))}$. These fitness score are used to calculate the **aggregate fitness indices**. The notation $\mathbf{FI(h_i,t)}$ denotes the *aggregate fitness index* of host h_i at time t. One fitness index function that we have found useful is: $\mathbf{FI(h_i,t) = (w_1 * PF(h_i,t)) + (w_2 * CF(h_i,t)) + (w_3 * NF(h_i,t))}$, where w_i is the weight given to the ith load metric, and $\Sigma\, w_i = 1.0$. The fitness index is a relative measure of host load: the higher the fitness index, the lighter the load on the host. When making resource allocation decisions, hosts with higher fitness scores are preferred over hosts with lower fitness scores.

5 Experimental Results

The techniques described in the previous sections have been implemented and employed to manage several subsystems within a complex distributed experimental Navy system. This section describes a set of experiments that demonstrate the ability of our approach to deliver real-time QoS to these subsystems, even in highly dynamic environments.

The experiments were performed in the System Control Laboratory (SCL) at the Naval Surface Warfare Center in Dahlgren, Virginia. The tactical system applications, simulation components, and resource and QoS management components were run on Sun, SGI, DEC and HP systems, consisting of (a) 12 Sun Ultra 2 Enterprise dual processor machines in a "compute farm" arrangement, (b) 2 Sun Ultra 2 dual processors, (c) 1 Sun Ultra 1, (d) 1 Sun Ultra 60, (e) 1 Sun Sparc 10, (f) 7 SGI Origin 200 dual processor machines in a compute farm arrangement, (g) 3 SGI Onyx 2 computers, (h) 1 SGI O2 workstation, and (i) 5 HP TAC-4 J210 workstations. In addition, there were 3 Sun systems running Solaris 2.5.1, two of which were used as file servers, and one of which was used as a Network Time Protocol (NTP) server.

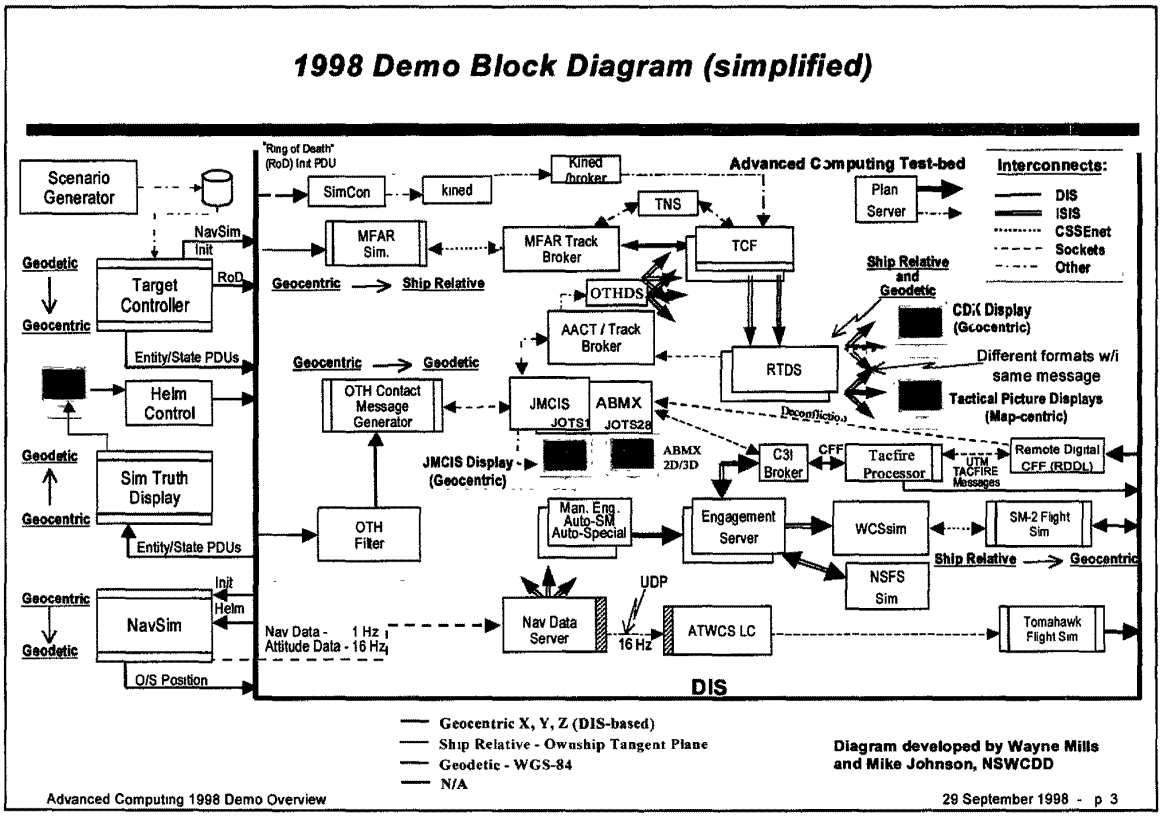


Figure 5. The software systems used in the experiment.

There were also 6 DEC Alpha workstations running DEC Unix 4.0b which were used for instrumentation displays, and 1 SGI Indigo2 running Irix 5.3 which was used for visualization of network loading. Four Windows NT systems were also part of the testbed but were not used during the experiments reported herein.

Three networks were used in the testbed: ATM, FDDI, and Ethernet (both 10baseT and 100baseT). All systems except for the file servers and time server were connected to the Ethernet network, and most hosts were also connected to either the ATM or FDDI networks. The ATM network was used as the primary network for tactical communication (except for hosts that did not have ATM connections, in which case FDDI was the preferred network). The Ethernet network was used primarily for data traffic supporting resource and QoS instrumentation, monitoring, and control.

The block diagram in Figure 5 shows the major computer program components that were operational during the experiment. The components shown outside of the shaded region represent simulators controlling targets, tracks, Navy ships, radars, and weapon systems. These components form an integrated wrap-around simulation environment capable of updating and controlling the behavior of 1000s of real-time tracks. The "system under test", represented within the shaded region, is composed of Command, Control, Communications, Computers, and Intelligence (C4I) components, ship combat systems Anti-Air Warfare (AAW) components, and Advanced Tomahawk Weapon Control System (ATWCS) Launch Control components.

The AAW subsystem was the focus of our experiments and incorporates the air defense path components discussed in Section 2: *threat detection* (detection & assessment path) and *engagement* (action initiation path). The data flows along the AAW subsystem paths are initiated by the simulation environment. Track data injected by the wrap-around simulation environment is provided to the Track Correlation and Filtering (TCF) function, which creates and maintains a track file representing the ship's view of the tactical environment. The TCF function provides this data to the Radar Track Data Server (RTDS), which distributes the track data to client applications as requested. The four weapons doctrine applications (Manual Engage, Semi Auto [not shown], Auto-SM, and Auto Special) are clients of the RTDS and compare the track data against operator selected doctrine criteria. They then forward any tracks that match the doctrine criteria to the Engagement Server for engagement. The Engagement Server validates and schedules the engagements and sends the engage order to the Weapons Control System Simulator (WCSsim). The WCSsim then sends the target information and firing order to the SM-2 missile flight simulator which launches and flies out a simulated missile which attempts to intercept and destroy the simulated target.

Within the AAW subsystem, the focus of our experiments is the AAW Auto Special Doctrine path, consisting of the Track Processor component of TCF, the RTDS application, the Auto Special Doctrine process, and the Engagement Server application. This path specified as AAW:Doctrine:AutoSpecial_Review in Figure 1.

Resource and QoS management components were used to monitor, control, and manage the testbed environment. Resource management components were used for starting up and configuring all of the system and simulation components (based on the QoS specifications described in Section 2) except for those shown within the heavy dashed line. Run-time monitoring of host, network and application resource usage and statuses were performed for all of the SGI, Sun, and HP systems within the testbed. In addition, for components within the AAW Doctrine subsystem paths (described above), application QoS performance and load metrics were monitored. This monitored information was used for determining host and network load and health indices, as well as application and path-level statuses and QoS performance metrics. These statuses, metrics, and load indices were used (as described in Section 4) for: (1) determining path overload conditions (i.e., QoS violations) and for determining the "best" method (from the options provided within the QoS specifications) for restoring required QoS and (2) determining fault recovery actions when software failures were detected. Detailed QoS specifications, as described in Section 2, were developed for each of the systems shown in the diagram (excluding those within the heavy dashed line). These specifications defined the startup, configuration, and reconfiguration options available for each software application and path. Software fault recovery actions were provided for ATWCS, resource management, displays, and AAW components during the experiments. Scalability options were provided for the AAW doctrine components (ManualEngage, SemiAuto, AutoSM, and AutoSpecial processes) within the various AAW doctrine paths.

A set of experiments was designed to test our approach to adaptive resource and QoS management for the class of applications described in [6]. As discussed above, the tests focused on the AAW subsystem, in particular on the AutoSpecial doctrine review path (the path specified in Figure 1 as AAW:Doctrine:AutoSpecial_Review). Hereafter the path is referred to as ASRP (AutoSpecial_Review Path). The path is described in the specification language as being scalable and survivable; thus, our experiments tested both of these features. Its real-time QoS requirements are maximum latency [λ_{REQ}(**ASRP**)] of 65ms with a sliding window size [ω(**ASRP**)] of 20 samples, and maximum number of violations during the sliding window [υ(

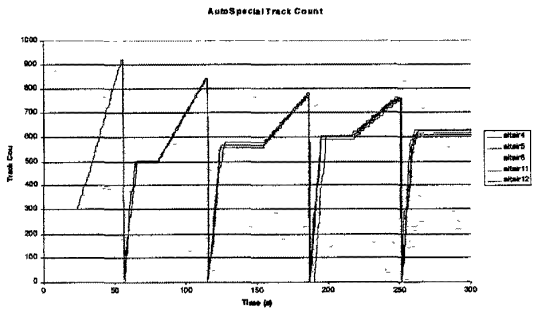


Figure 6. Tactical Load |ASRP.DS| over time.

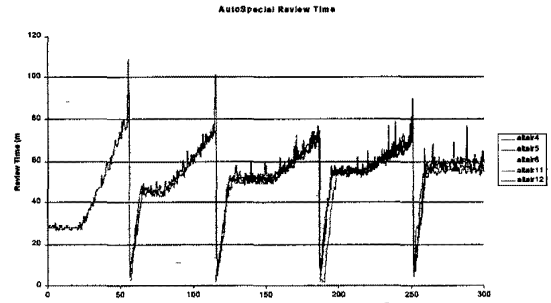


Figure 7. Review Latency λ_{OBS}(ASRP) as a function of *time*.

ASRP)] of 15 samples. The system is stressed with dynamic changes in the workload of the path (e.g., increasing track load), resulting in QoS violations. We test the ability of the QoS and resource management middleware to detect and recover from the violations, and to do so in a timely manner. Additionally, a set of experiments is run to test the ability of the middleware to provide survivability (fault detection and recovery) services to real-time application systems in a timely manner. The results are summarized in Figures 6 and 7.

To test the ability of the middleware to manage real-time QoS in dynamic environments, the workload of the path was incrementally increased (see Figure 6) until its timing requirement could no longer be met (see Figure 7). At that point, the scalability feature of the path was exploited - the QoS and resource management software replicated a program contained in ASRP and assigned the replica to the host with the highest fitness.

Application AAW:Doctrine:AutoSpecial (abbreviated as AS) is the one that requires replication in order to restore real-time QoS to path ASRP during the test runs. The eligible list of hosts where the AS application could be run was **Profile(AS)** = {altair1, altair4, altair5, altair6, altair11 and altair12}. Initially, only one copy of the AS application was run on host altair4. Hence, **REPLICAS(AS)** = $\{AS_1\}$ (indicated by a single line representing the application on host altair4 in Figure 6) and the tactical load of the path **|ASRP.DS|** = **|ASRP.DS(AS)|** = 300 (also as shown in Figure 6). As seen in figure 7, the initial review time latency is less than the required review time latency of 65ms (λ_{OBS}**(ASRP)** < λ_{REQ}**(ASRP)**).

Starting at about time 25, the tactical load **|ASRP.DS|** is increased gradually until at about time 50, **|ASRP.DS|** > 900 and λ_{OBS}**(ASRP)** > λ_{REQ}**(ASRP).** At time 53.27 (see Figure 7), the QoS monitor detects that at least υ**(ASRP)** violations occurred within the sampling window ω**(ASRP)**, and the QoS management middleware reports a real-time QoS violation. The resource management software is notified that the AS application should be replicated to recover from the violation. Using the host fitness indices for all hosts in **Profile(AS)**, the host on which to start the replica is selected (which in this case was host altair11). The host fitness index function used is $\mathbf{FI(h_i,t) = (w_1 * PF(h_i,t)) + (w_2 * CF(h_i,t)) + (w_3 * NF(h_i,t))}$, as described in Section 4. The time for the resource management response took 0.0009 seconds, and the total latency of the recovery actions (including starting the new replica of AS) took 0.3017 seconds.

Following the recovery action, **REPLICAS (AS)** = $\{AS_1, AS_2\}$ (indicated by two lines representing hosts altair4 and altair11 in Figure 6). The tactical load of each replica stabilizes at about time 60, such that $\mathbf{|ASRP.DS(AS_1)|} = \mathbf{|ASRP.DS(AS_2)|} = 500$ (thus the path's total tactical load **|ASRP.DS|** = 1000). As seen in Figure 7, the observed latency for each replica stabilizes below the required latency (λ_{OBS}**(ASRP)** < λ_{REQ}**(ASRP)**).

This sequence of dynamically increasing the tactical load **|ASRP.DS|** until the ASRP path is overloaded is repeated several more times, (starting at about time 70) as can be observed in Figures 6 and 7. The results are very similar to the first case-the QoS violation is quickly detected and a scale-up action is taken. Across all the tests, the average resource allocation decision time was 0.0012 seconds (with a standard deviation of 0.0002582) and the average total latency of the recovery actions was 0.32951667 seconds (with a standard deviation of 0.04376704). These results show the effectiveness of our approach for dynamic real-time QoS management within a large-scale combat system as described in [6]. Real-time QoS violations were successfully detected, appropriate recovery actions were performed, and the required QoS was restored. These actions were consistently performed in less than one second, even under heavy tactical and system loads.

In addition to testing the ability of our middleware to deliver real-time QoS to dynamic real-time applications, we also tested the ability of the middleware to provide survivability services to real-time application systems in a timely manner. In these tests, one replica of the AutoSpecial application was faulted, requiring that the middleware (1) detect the failure and (2) restart a replica on the "fittest" of the eligible hosts. These tests were performed a total of 17 times, and the reallocation decision times and total recovery times were measured. The average resource allocation decision time was 0.00097059 seconds, with a standard deviation of 0.00041648. The minimum, average and maximum total

latencies of the recovery actions were 0.1296, 0.19401765, and 0.2379 seconds, respectively, with a standard deviation of 0.04376704. Thus, across all tests, the total response time for application fault detection and recovery services was far less than one second, providing adequate response times.

6 Conclusions and Future Work

This paper has presented adaptive QoS and resource management technology for distributed real-time systems with *a-posteriori*-determined workloads. The effectiveness of our approach was demonstrated within an experimental large-scale combat system. In particular, test results were obtained and evaluated for the AutoSpecial_Review path of the AAW subsystem. The results show that real-time QoS violations and program faults were successfully detected, appropriate recovery actions were performed, and the required QoS was restored. These actions were consistently performed in less than one second, even under heavy tactical and system loads. Ongoing work includes formal techniques and decentralized algorithms for QoS negotiation, resource and QoS management for transient and quasi-continuous paths, and benchmarking of dynamic real-time systems.

7 References

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

[2] 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.

[3] T.P. Baker, "Stack-based scheduling of realtime processes," *Journal of Real-time Systems*, **3(1)**, March 1991, 67-99.

[4] 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.

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

[6] 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.

[7] 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.

[8]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.

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

[10] 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.

[11] 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.

[12] K. Ramamritham, J.A. Stankovic and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Transactions on Computers,* **38(8)**, August 1989, 110-123.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] J. Verhoosel, L. R. Welch, D. K. Hammer, and E. J. Luit, "Incorporating temporal considerations during assignment and pre-run-time scheduling of objects and processes," *Journal of Parallel and Distributed Computing, 36(1)*, July 1996, 13-31, Academic Press.

[19] 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.

[20] 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.

[21] J. Xu and D.L. Parnas, "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Transactions on Software Engineering,* **16(3)**, March 1990, 360-369.

Ex.1009 / Page 269 of 280
TESLA, INC.

APPENDIX G

RESOURCE MANAGEMENT
QoS AND SYSTEM SPECIFICATIONS

Larry A. Fontenot+, Lonnie R. Welch*, Shafqat Anwar*, Paul V. Werme+

+Naval Surface Warfare Center, Dahlgren Division
17320 Dahlgren Road
Dahlgren, Virginia 22448-5000
{FontenotLA, WermePV}@nswc.navy.mil

*Computer Science and Engineering Dept
The University of Texas at Arlington
Arlington, TX 76019-0015
Welch@cse.uta.edu
Anwar@swbell.net

APPENDIX G

RESOURCE MANAGEMENT
QoS AND SYSTEM SPECIFICATIONS

Larry A. Fontenot+, Lonnie R. Welch*, Paul V. Werme+

+Naval Surface Warfare Center, Dahlgren Division
17320 Dahlgren Road
Dahlgren, Virginia 22448-5000
{FontenotLA, WermePV}@nswc.navy.mil

*Computer Science and Engineering Dept
The University of Texas at Arlington
Arlington, TX 76019-0015
Welch@cse.uta.edu

G.1 QoS and System Specifications.

a. To effectively manage a pool of computing resources, the Resource Manager must have some means of determining the capabilities and configuration of the computing resources under its control, of determining the software components that need to be executed and the dependencies of these software components on both hardware and software resources, determining what mission-level and application-level requirements are expected to be met, and determining what control capabilities are available to be used to attempt to recover from fault or QoS violation conditions. To address these needs, a System and Software Specification Grammar has been developed to attempt to capture the "static" information needed by the Resource Manager for effectively managing a pool of distributed resources. The development of this grammar has been a joint effort between NSWCDD and the University of Texas at Arlington (UTA). The grammar attempts to capture the follow information:

(1) Hardware and Operating Systems
- Hardware Configuration
- Network Configuration
- Operating System and Version

(2) Software
- Systems, Subsystems, Applications, Processes
- Resource Requirements
- QoS Requirements
- Survivability Requirements
- Path Information: Structure and QoS Requirements

b. As part of the grammar development effort, a specification library has also been developed which parses the specification files and provides an API for accessing the

specification information. The specification library was written in C and has been ported for all of the UNIX development platforms in the testbed, including Solaris 2.6, Solaris 2.5.1, Irix 5.3, Irix 6.3, Irix 6.4, and HP-UX 10.20. The library is currently being used by most of the Resource Management components, including Program Control, Resource Manager, Path QoS Managers, History Servers, UNIX Host Monitors, and the Host and Path Displays.

c. The remainder of this section presents the Resource Management System and Software Specifications Grammar for describing the characteristics and requirements of dynamic, path-based real-time systems. The grammar provides abstractions to describe the properties of the software, such as hierarchical structure, inter-connectivity relationships, and run-time execution constraints. It also allows description of the physical structure or composition of the hardware such as LANs, hosts, interconnecting devices or ICs (such as bridges, hubs, and routers), and their statically known properties (e.g., peak capacities). Further, the Quality-of-Service (QoS) requirements on various system components can be described.

d. At the highest level, a specification consists of a collection of software systems, hardware systems, and network systems. The language rules for specifying systems are described in the remainder of this section. A high-level system specification is shown below:

```
SOFTWARE SYSTEM AAW {...}
SOFTWARE SYSTEM ATWCS {...}
HARDWARE SYSTEM HOST_POOL {...}
NETWORK SYSTEM SHIP_NET {...}
```

G.1.1 Software System Specifications.

A software specification is collection of software systems, each of which consists of one or more software subsystems. This is illustrated below:

```
SOFTWARE SYSTEM AAW {
        SUBSYSTEM Displays {...}
        SUBSYSTEM Tactical_Services {...}
        SUBSYSTEM Doctrine {...}
        ...
} // End Software System AAW

SOFTWARE SYSTEM ATWCS {
        SUBSYSTEM Launch_Control {...}
        SUBSYSTEM Engagement_Plan_and_Control {...}
        SUBSYSTEM Mission_Data_Plan {...}
        SUBSYSTEM Scenario_Gen {...}
        ...
} // End Software System ATWCS
```

Note that a comment begins with "//" and extends to the end of a line.

Qualified references: AAW:Displays, AAW:Tactical_Services, and AAW:Doctrine denote the subsystems of software system AAW. This is distinguished from the subsystems of ATWCS, which would be identified as ATWCS:Launch_Control,

ATWCS:Engagement_Plan_and_Control, ATWCS:Mission_Data_Plan and ATWCS:Scenario_Gen.

G.1.1.1 Software Subsystem Specifications.

A subsystem is specified by describing its priority, sets of constituent applications and devices, a set of end-to-end real-time path definitions, and a graph representing the communication connectivity of the applications and devices. A sample subsystem specification is shown below:

```
SUBSYSTEM Doctrine {
        Priority 2;
        PATH Spy_Declared_AutoSpecial {...}
        PATH AutoSpecial_ReviewTime {...}
        PATH SemiAuto_ReviewTime {...}
        PATH AutoSM_ReviewTime {...}
        ...
        APPLICATION Auto_Special {...}
        APPLICATION Semi_Auto {...}
        APPLICATION Auto_SM {...}
        ...
} // End SubSystem Doctrine
```

G.1.1.1.1 Dynamic Real-time Path Specifications.

a. The definition of a path includes a set of constituent applications, various path attributes, QoS requirements, and data/event stream definitions (see example below). The attributes of a path include priority, type, and importance. Path type, which defines the execution behavior of the path, is either continuous, transient, or quasi_continuous.

b. A continuous path is one in which the elements of a data stream are continuously evaluated and decisions are continuously made whether or not any of the elements require action. Typically, there is a timeliness objective associated with completion of one review cycle, i.e., on the time to review each of the elements of the data stream once. The doctrine track-review path from track-distribution to doctrine evaluation is an example of a data-stream that is constantly undergoing analysis.

c. Transient paths are typically event-driven. An action in the system initiates a task to be performed to completion. A timing objective is typically associated from initiation to task completion. A good example of a transient path is the Spy-Declared Auto-Special Engagement. Spy initiates the action to send a track through a high-priority path with tight timing requirements.

d. Finally there is the quasi-continuous path. This typically occurs when an action "turns-on" a path and a later action "turns-off" the path. There are typically two timeliness objectives: (1) completion time for one cycle and (2) deactivation time; typically, it is more critical to perform the required processing before the activation deadline than it is to meet the completion time for each cycle. Thus, it is acceptable for the completion time of some cycles to

violate the requirements, as long as the desired actions are completed by the deactivation deadline.

```
PATH AutoSpecial_ReviewTime {
        Connectivity{...}
        Type Continuous;
        Priority 1;
        RealTimeQoS{...}
        Scalability{...}
        DATASTREAM{...}
} // End Path AutoSpecial_ReviewTime
```

G.1.1.1.1.1 Path Connectivity Graphs.

The connectivity specification represents the communication relationships among applications in a path. These relationships form a graph, which is specified as a set of ordered application pairs. The sample specification (below) indicates that application AAW:Track_Control:Track_Controller sends data to application AAW:Track_Distribution:RTDS and that application AAW:Track_Distribution:RTDS sends data to application AAW:Doctrine:Auto_Special. Note that the names of applications are fully qualified, as System:Subsystem:Application.

```
Connectivity {
        (AAW:Track_Control:Track_Controller, AAW:Track_Distribution:RTDS);
        (AAW:Track_Distribution:RTDS, AAW:Doctrine:Auto_Special);
} // End Connectivity
```

G.1.1.1.1.2 Real-time QoS.

```
RealTimeQoS {
        SimpleDeadline 65.0;            // Cycle deadline
        InterProcessingTime 0.600;
        Throughput 200;
        MaxSlack 80;                    // Maximum deadline slack/cycle
        MinSlack 20;                    // Minimum deadline slack/cycle
        SlidingWindowSize 20;           // Cycles to monitor real-time QoS
        Violations 15;                  // Max QoS violations w/in window
} // End RealTimeQoS
```

As seen in the above example, a real-time QoS specification includes timing constraints such as simple deadlines, inter-processing times, and throughputs. A simple deadline is defined as the maximum end-to-end path latency during a cycle of a continuous or quasi-continuous path, or during an activation of a transient. Inter-processing time is defined as a maximum allowable time between processing of a particular element of a continuous or quasi-continuous path's data stream in successive cycles. The throughput requirement is defined as the minimum number of data items that the path must process during a unit period of time. Each timing constraint specification may also include items that relate to the dynamic monitoring of the constraint. These include minimum and maximum slack values (that must be maintained at run-time), the size of a moving window of measured samples that should be observed, and the maximum tolerable number of violations (within the window).

G.1.1.1.1.3 Scalability.

The grammar and model consider the scalability of the end-to-end paths and their application program constituents. Some paths permit replication of their constituent applications to *scale* to dynamic data stream or event stream loads. If a scalable path is unable to meet its real-time requirements, one or more of its constituent applications may be replicated. Similarly, if a path is exceeding its real-time requirements by a large margin, one or more of its replicas may be removed. The scalability specification contains a flag that is TRUE or FALSE, indicating if the path is scalable. Also specified is the PathSettlingTime, which indicates the amount of time that must be allowed between successive "scalings" of the path. Below is an example of a scalability specification.

```
Scalability {
        Scalable TRUE;            // Path has scalable components
        PathSettlingTime 40.00;   // Seconds between reconfigurations
} // End Scalability
```

G.1.1.1.1.4 Datastream Specification.

a. For real-time systems it is important to understand the characteristics of the environment in which they operate. We have found it useful from an engineering perspective to model an environment as deterministic, stochastic or dynamic. A deterministic environment exhibits behavior that can be characterized by a constant value (scalar or interval). A stochastic environment behaves in a manner that can be characterized by a statistical distribution, which may be either a well-known distribution, e.g. normal, or an empirical distribution that has properties that can be derived from a data set. A dynamic environment depends on conditions that cannot be known in advance. For systems that operate in such environments, it can be catastrophic to build systems based on predicted conditions because the system may not be able to adapt, even though adequate resources may be available.

```
DATASTREAM {
        Type Dynamic;             // Deterministic, Stochastic, or Dynamic
        SlackQoS 400;             // Additional number of data items that the
                                  // continuous/quasicontinous path should be
                                  // able to handle at any given time
} // End Datastream
```

b. The data stream size or event arrival rate of a dynamic stream is not described in the specification, since it must be observed at run time.

c. The SlackQoS specification for a datastream indicates an amount of additional elements that the path should be able to process in a timely manner. The resource allocator should consider this quantity when assessing possible allocations.

G.1.1.1.2 Application Specifications.

```
APPLICATION Auto_Special {
```

```
        TimeDelay 4;            // Delay this long after initial startup
        Automatic TRUE;         // Automatically start this app.
        Console FALSE; // Start this App in an XTerm or other console
        RM_Start TRUE; // Should allocations be done dynamically
        STARTUP{...}
        SHUTDOWN{...}
        DEPENDENCY{...}
        RestartDelay 10;
        SurvivabilityQoS {...}
        Scalability {...}
} // End Application Auto_Special
```

An application is an executable image that may be started as an autonomous process on a host. *TimeDelay* indicates the amount of time that must elapse since the startup of the previous application (this is sometimes needed to insure proper initialization of cooperating applications). *Automatic* indicates whether the application is part of a default startup configuration for the system or not (necessary for one-button system starts). Application attributes also include all information necessary to startup and shutdown applications (not elaborated in this paper). *Console* indicates whether or not an application is to be started in an Xterm, or some other type of console. *RM_Start* specifies whether or not the application is to be dynamically allocated at startup, or rely on an operator for resource allocation. The *startup* block and the *shutdown* block describe how to automatically start and stop the application. The *dependency* block indicates any dependencies the application may have with the startup and/or shutdown of other applications (e.g. it may be required that a particular application be started before another application can be started). *RestartDelay* indicates the amount of time that must elapse before restarting a "failed" application. The *SurvivabilityQoS* and *Scalability* blocks indicate if and how survivability and scalability services are to be provided to the application.

G.1.1.1.2.1 Application Startup Information.

An application *startup* block contains all the information necessary to, automatically or manually, start an application. This information includes supported hardware (host) type, operating-system type, and operating-system version(s) (see example below). This may be further constrained by an optional list of the names of hosts that can run the application. The startup information also includes the working directory for reading and writing data files, the name of the executable, and an *ordered* list of arguments that must be passed on the command line when the application is started. Last is a list of processes expected to be seen on the system when the application is running.

```
STARTUP {
        Type "SUN";
        OS "Solaris";
        Version "2.5.1";
        Version "2.6";
        Host altair1;
        Host altair4;
        Host altair5;
        Host altair6;
        Host altair7;
        Host altair8;
```

```
        Host altair11;
        Host aquilla;
        Host blofeld;
        Directory "$HIPERD_AAW_VERSION/exes";
        Execute "auto_special.solaris2.6.exe";
        Arg "%(UNIQUE, 1, 32)";
        Arg "A_Spcl_%(UNIQUE, 1, 32)";
        Arg "-jewel";
        Arg "-rstat";
        Arg "-splot";
        Arg "> /usr/tmp/%(USERID)_auto_special%(UNIQUE, 1, 32).out";
        PROCESS auto_special.solaris2.6.exe {}
    } // End Startup
```

G.1.1.1.2.2 Application Shutdown Information.

An application shutdown block indicates the command(s) to be used for termination of the application. A shutdown command may be a POSIX signal name or may be a shell script or batch file. A sample shutdown block is shown below.

```
SHUTDOWN {
        Signal "SIGTERM";
} // End Shutdown

SHUTDOWN {
        Script "$HIPERD_AAW_VERSION/exes/stop_auto_special.sh";
} // End Shutdown
```

G.1.1.1.3 Inter-application Dependencies.

```
DEPENDENCY {
        Type STARTUP;
        Name "AAW:Displays:State_Server";
        Delay 5;                                // Seconds
}
```

A *dependency* block describes a temporal relationship between applications (see above example). The relationship indicates the *type* of the dependency (startup or shutdown), the name of the program with which the dependency exists, and the time value associated with the relationship. The time value indicates the duration that must elapse between start or stop of the named application and the start or stop of the application which has the dependency block in its specification.

G.1.1.1.4 Application Survivability QoS.

```
SurvivabilityQoS {
        Survivable TRUE;        // Application is survivable
        MinCopies 1;            // Min replicas of this application
        SameHost FALSE;         // Restart on same host on failure?
} // SurvivabilityQoS
```

As shown in the above example, a survivability QoS specification includes a boolean variable that indicates (1) whether the application should be managed to ensure survivability and (2) the minimum required level of redundancy. *SameHost* allows an application to be specified for restart only on the host it was running upon failure.

G.1.1.1.5 Application Scalability.

```
Scalability {
        Scalable TRUE;          // Is app scalable?
        Combining TRUE;         // Does combine inputs
        Splitting TRUE;         // Does divide outputs
} // Scalability
```

The *scalability* specification for an application indicates if an application can be scaled via replication (see example above). Scalable applications are programmed to exploit load sharing among replicas, and can adapt dynamically to varying numbers of replicas. The specification also indicates whether an application combines its input stream (which may be received from different predecessor applications and/or devices), and splits its output stream (which may be distributed to different successor applications and/or devices) are also specified. "Combining" and "splitting" are commonly called "forking" and "joining" in parallel computing paradigms.

G.1.1.2 Example.

The completed software example for an Auto-Special doctrine application would be constructed as follows:

```
SOFTWARE SYSTEM AAW {
        SUBSYSTEM Doctrine {
                Priority 2;
                PATH AutoSpecial_ReviewTime {
                        Connectivity {
                                (AAW:Track_Control:Track_Controller, AAW:Track_Distribution:RTDS);
                                (AAW:Track_Distribution:RTDS, AAW:Doctrine:Auto_Special);
                        } // End Connectivity
                        Type Continuous;
                        Priority 1;
                        RealTimeQoS {
                                SimpleDeadline 65.0;
                                InterProcessingTime 0.600;
                                Throughput 200;
                                MaxSlack 80;
                                MinSlack 20;
                                SlidingWindowSize 20;
                                Violations 15;
                        } // End RealTimeQoS
                        Scalability  {
                                Scalable TRUE;
                                PathSettlingTime 40.00;
                        } // End Scalability
                        DATASTREAM {
```

```
                Type Dynamic;
                SlackQoS 400;
            } // End DataStream
        } // End Path AutoSpecial_ReviewTime
        APPLICATION Auto_Special {
            TimeDelay 4;
            Automatic TRUE;
            Console FALSE;
            RM_Start TRUE;
            STARTUP {
                Type "SUN";
                OS "Solaris";
                Version "2.5.1";
                Version "2.6";
                Host altair1;
                Host altair4;
                Host altair5;
                Host altair6;
                Host altair7;
                Host altair8;
                Host altair11;
                Host aquilla;
                Host blofeld;
                Directory "$HIPERD_AAW_VERSION/exes";
                Execute "auto_special.solaris2.6.exe";
                Arg "%(UNIQUE, 1, 32)";
                Arg "A_Spcl_%(UNIQUE, 1, 32)";
                Arg "-jewel";
                Arg "-rstat";
                Arg "-splot";
                Arg "> /usr/tmp/%(USERID)_auto_special%(UNIQUE, 1, 32).out";
                PROCESS auto_special.solaris2.6.exe {}
            } // End Startup
            SHUTDOWN {
                Signal "SIGTERM";
            } // End Shutdown
            DEPENDENCY {
                Type STARTUP;
                Name "AAW:Displays:State_Server";
                Delay 5; //secs
            } // End Dependency
            RestartDelay 10;
            SurvivabilityQoS {
                Survivable TRUE;
                MinCopies 1;
                SameHost FALSE;
            } // End SurvivabilityQoS
            Scalability {
                Scalable TRUE;
                Combining FALSE;
                Splitting FALSE;
            } // End Scalability
        } // End Application Auto_Special
    } // End SubSystem Doctrine
} // End System AAW
```

G.1.2 Hardware System Specifications.

A hardware system specification construct allows the description of one or more hardware subsystems (see example below). Each hardware subsystem consists of one or more hosts. A host specification describes the host's name, type, operating system version, speed, RAM capacity, CPU quantity, and network connections.

```
HARDWARE SYSTEM HOST_POOL {
        HOST altair1 {
                Type "SUN";
                OS " Solaris";
                Version "2.6";
                Speed 200;      // MHz
                Memory 128;     // MB
                NumCPUs 1;
                Default-Network SHIP_NET:ATM;
                Network SHIP_NET:ETHER;
        }
        HOST altair2 {...}
        HOST altair3 {...}
        HOST altair4 {...}
} // End Hardware System HOST_POOL
```

G.1.3 Network System Specifications.

A network system specification describes the LANs and ICs (interconnection devices such as switches, hubs and routers). A system consists of one or more subsystems. A subsystem may contain LANs (each with an associated peak bandwidth specification) and ICs (each containing a description of network membership). A sample network specification is shown below:

```
NETWORK SYSTEM SHIP_NET {
        LAN ATM {
                Bandwidth 100;
        }
        LAN ETHER {
                Bandwidth 100;
        }
        IC Router {
                Network SHIP_NET:ATM;
                Network SHIP_NET:ETHER;
        }
} // End Network System Ship_Net
```

G.1.4 Summary.

This section has described a specification grammar for declaring requirements on applications in a dynamic, distributed, heterogeneous resource pool. The grammar allows the description of environment-dependent application features, which allows for the modeling and dynamic resource management of such systems.