AD-A227 972

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1990	~~THESIS~~/DISSERTATION

4. TITLE AND SUBTITLE
Horizontal Fault Tolerance in a Fully Distributed Loosely Coupled Environment

5. FUNDING NUMBERS

6. AUTHOR(S)
Peter Schiavi

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Student Attending: Texas A&M Univ

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA- 90-030D

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFIT/CI
Wright-Patterson AFB OH 45433-6583

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release IAW 190-1
Distributed Unlimited
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

DTIC
ELECTE
NOV 0 2 1990
S B D

DTIC FILE COPY

14. SUBJECT TERMS

15. NUMBER OF PAGES
338

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1 Agency Use Only *(Leave blank)*

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s) Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes Enter information not included elsewhere such as. Prepared in cooperation with..., Trans. of..., To be published in ... When a report is revised, include a statement whether the new report supersedes or supplements the older report

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

HORIZONTAL FAULT TOLERANCE IN A FULLY DISTRIBUTED LOOSELY COUPLED ENVIRONMENT

A Dissertation

by

PETER SCHIAVI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 1990

Major Subject: Computer Science

HORIZONTAL FAULT TOLERANCE IN A FULLY DISTRIBUTED LOOSELY COUPLED ENVIRONMENT

DTIC
COPY
INSPECTED
6

A Dissertation

by

PETER SCHIAVI

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Approved as to style and content by:

Udo W. Pooch
(Chair of Committee)

Donald K. Friesen
(Member)

Jeffrey R. Miller
(Member)

William M. Lively
(Member)

Richard A. Volz
(Head of Department)

December 1990

ABSTRACT

Horizontal Fault Tolerance in a Fully Distributed
Loosely Coupled Environment. (August 1990)
Peter Schiavi, B.S., Case Institute of Technology;
M.C.I.S, Cleveland State University
Chair of Advisory Committee: Dr. Udo W. Pooch

The increasing use of local area networks to divide up the processing power once allocated to a single central processor has a side benefit which allows for the implementation of levels of fault tolerance at minimal cost. With a central processor, hardware replication is mandatory to continue processing in the face of hardware failures. Otherwise, processing must generally halt for a period of hardware repair. A local area network already contains replicated hardware, along with software to support communications over links connecting the individual nodes.

The existence of duplicated hardware, and independent processing ability within each node, allows for a concentration on *software* support for fault tolerance. Hardware replication can be limited to such areas as network topologies which employ multiple links among the nodes.

This research concentrates on software approaches to fault tolerance in a (loosely coupled) network environment. Current approaches are studied. These turn out to emphasize special purpose languages and operating systems designed to allow for *transparent* distribution of tasks amongst the nodes. The failure scenarios under which faults will be masked varies widely.

next page

Specifically, this research develops a set of language and operating system *protocols* to implement a level of fault tolerance. This *Fault Tolerant Monitor* (FTM) system is layered above the operating system. (KR) Cooperating applications which desire to avail themselves of the fault tolerance services offered by the FTM system *advertise* themselves to it. Other applications may simultaneously be executing on the network as if the FTM system were not present.

The FTM system consists of an FTM module running on each node. These communicate together. Applications are provided with an *Intercept Library* which is linked into each executable. This defined procedure interface isolates the applications from the FTM system protocols. Services such as node-independent message delivery and restart/relocation of failing applications are offered.

An FTM system following the protocols is developed and implemented under UNIX. It is utilized to generate performance statistics, and performance is compared between applications which utilize the FTM services, and those which bypass them.

To my parents for their long-standing encouragement,
and to my nephew and nieces for serving me with a reminder
that there is a reason to help make things better.

– and –

To anyone and everyone who ignored my irrational
habits during this student period at Texas A&M.

ACKNOWLEDGMENTS

The author's graduate studies at Texas A&M University were funded by the United States Air Force through the Civilian Institution program administered by the Air Force Institute of Technology (AFIT) at Wright-Patterson AFB, Ohio.

My deepest and most sincere gratitude is extended to Dr. Udo W. Pooch, my chairman, advisor, motivator, and friend. His professional expertise, along with his ability to channel his knowledge on a personal basis, were instrumental and irreplaceable. Without Dr. Pooch's constant support, successful completion of this research would not have been possible. Thanks also go to Dr. Pooch's large family, and even larger accumulation of semi-reliable automobiles, for providing me with something to do, and a homey atmosphere to do it in, during those periods when a break from school was possible.

Sincere appreciation is extended to Dr. Donald K. Friesen and Dr. William M. Lively, Computer Science, and Dr. Jeffrey R. Miller, Accounting, for their support, suggestions, and encouragement. Special thanks is also due to Dr. S. Charles Maurice, Economics, for agreeing to serve as the graduate council representative.

Recognition is extended to Dave Cook for his aid in using various MacIntosh utilities to generate many of the figures.

TABLE OF CONTENTS

	Page
ABSTRACT	iii
DEDICATION	v
ACKNOWLEDGMENTS	vi
TABLE OF CONTENTS	vii
LIST OF TABLES	xi
LIST OF FIGURES	xii
CHAPTER	
I INTRODUCTION	1
1.1 Motivation	1
1.2 Approach	2
1.3 Overview	2
II REVIEW OF LITERATURE	4
2.1 Introduction	4
2.1.1 Classifications	4
2.1.2 Survey Articles	6
2.1.3 Mathematical Classifications	6
2.2 Hardware Fault Tolerance	6
2.2.1 System Level Considerations	6
2.2.2 Approaches Using Additional Hardware	8
2.2.2.1 Dual Components	8
2.2.2.2 Triple Modular Redundancy	9
2.2.2.3 Watchdog Processors	10
2.2.2.4 Design Diversity	10
2.2.3 Hardware Fault Tolerant Systems	11
2.2.3.1 Chip Level Considerations	12
2.2.3.2 Intel 432	12
2.2.3.3 Self-Testing and Repairing (STAR) Computer	14
2.2.3.4 Fault Tolerant Multiprocessor	17
2.3 Software Fault Tolerance	22
2.3.1 Software Approaches	22

CHAPTER Page

2.3.2 Software Fault Tolerant Systems 24
2.3.2.1 PLURIBUS 25
2.3.2.2 Software Implemented Fault Tolerance (SIFT) 31
2.3.2.3 UNIX RTR 34
2.3.2.4 MACH 37
2.4 Distributed Systems 41
2.4.1 Distributed Software 42
2.4.2 Backup, Checkpoint, Recovery 44
2.4.3 Consensus 47
2.4.4 Clock Synchronization 48
2.4.5 Distributed Operating Systems 50
2.4.5.1 LOCUS 52
2.4.5.2 MEDUSA 60
2.4.6 Distributed Programming Languages 67
2.4.6.1 PLITS 69
2.4.6.2 ARGUS 72
2.4.6.3 *MOD 81
2.4.6.4 Synchronizing Resources (SR) 85
2.5 Fault Tolerant Networks 91
2.5.1 Configuration Reliability 92
2.5.2 Routing and the ARPANET 94
2.5.3 Autonomous Decentralized Systems 96
2.5.3.1 Planar-2 Network Configuration 97
2.5.3.2 Advanced On-Board Signal Processor (AOSP) 98

III FTM DESIGN PHILOSOPHY 104

3.1 Introduction 104
3.2 Hardware Environment 105
3.3 Software Environment 105
3.4 Layered Fault Tolerance 106
3.5 Layered Distributed System FT Characteristics 106
3.6 Fault Tolerant Monitor (FTM) 107
3.7 Intercept Code (IC) 108
3.8 Overview of Services Offered 109
3.8.1 Connect to FTM System 110
3.8.2 Message Delivery 111
3.8.3 Task Relocation 111
3.8.4 Critical Files 111
3.8.5 Status 112
3.8.6 Disconnect from FTM System 112

CHAPTER Page

3.9 Introduction to User Interface 112
3.9.1 Globally Advertise a Name 113
3.9.2 Receive a Message 114
3.9.3 Send a Message 115
3.9.4 Disconnect from FTM System 115
3.10 Protocols 116

IV FTM IMPLEMENTATION 119

4.1 Introduction 119
4.2 Implementation Overview 119
4.3 Software Choices 120
4.4 Hardware Choices 120
4.4.1 Disk Mirroring 121
4.5 FTM Library 123
4.5.1 File: *Readme* 124
4.5.2 File: *Makefile* 124
4.5.3 File: *header.h* 125
4.5.3.1 Global Structure Definitions 125
4.5.4 File: *ftms.lis* 126
4.5.5 File: *bkup.lis* 126
4.5.6 FTM System Source Files 126
4.5.7 Test Program Source Files 127
4.6 Multiple FTMs 127
4.7 FTM System Startup 127
4.8 FTM System Steady State Operation 128
4.9 FTM Logic Flow 128
4.9.1 Connect to other FTMs 129
4.9.2 Accept IC Connections 129
4.9.3 Read Messages from other FTMs 131
4.9.4 Read Messages from Connected ICs 132
4.9.5 Handle Pending Messages 132
4.9.6 Loop Control 138
4.10 IC Logic Flow 139
4.11 Example of an FTM Session 141

V RESULTS 145

5.1 Introduction 145
5.2 Feature by Feature Comparison 145
5.3 Validation of Implementation 148
5.4 Baseline Performance/Reliability 149
5.4.1 Advertise a Name (*adv*) 151

CHAPTER Page

5.4.2 Disconnect from FTM (*ftmclose*) 151
5.4.3 Obtain FTM Status Information (*ftmstatus*) 153
5.4.4 Obtain Node of an Application (*ftmwhere*) 153
5.4.5 Send a User Message (*to*) 154
5.4.6 Critical File Backup (*copycf*) 160
5.4.7 Read a User Message (*from*) 161
5.5 Automatic Relocation of Failing Applications 161
5.5.1 Relocation Statistics 162
5.6 Timing Statistics Summary 164
5.7 Reliability Matters 166

VI CONCLUSIONS AND RECOMMENDATIONS 170

6.1 Introduction 170
6.2 Conclusions 170
6.3 Recommendations 171

REFERENCES 177

APPENDICES

A FTM SYSTEM PROTOCOLS 187
B FTM PROTOTYPE DISTRIBUTION LIBRARY 202
C PERFORMANCE AND VALIDATION TESTING PROGRAMS FOR THE FTM PROTOTYPE 309

VITA 338

CHAPTER Page

5.4.2 Disconnect from FTM (*ftmclose*) 151
5.4.3 Obtain FTM Status Information (*ftmstatus*) 153
5.4.4 Obtain Node of an Application (*ftmwhere*) 153
5.4.5 Send a User Message (*to*) 154
5.4.6 Critical File Backup (*copycf*) 160
5.4.7 Read a User Message (*from*) 161
5.5 Automatic Relocation of Failing Applications 161
5.5.1 Relocation Statistics 162
5.6 Timing Statistics Summary 164
5.7 Reliability Matters 166

VI CONCLUSIONS AND RECOMMENDATIONS 170

6.1 Introduction 170
6.2 Conclusions 170
6.3 Recommendations 171

REFERENCES 177

APPENDICES

A FTM SYSTEM PROTOCOLS 187
B FTM PROTOTYPE DISTRIBUTION LIBRARY 202
C PERFORMANCE AND VALIDATION TESTING PROGRAMS FOR THE FTM PROTOTYPE 309

VITA 338

LIST OF TABLES

Table		Page
I.	IC-to-FTM, FTM-to-FTM, and FTM-to-IC Protocols	118
II.	FTM Message Handling Routines	133
III.	Timing Data: *adv*	152
IV.	Timing Data: *ftmclose*	152
V.	Timing Data: *ftmstatus*	153
VI.	Timing Data: *ftmwhere*	154
VII.	Timing Data: *to* (Single Node)	156
VIII.	Timing Data: *to* (Separate Nodes)	158
IX.	Timing Data: UNIX Socket (Single Node)	159
X.	Timing Data: INET Socket (Separate Nodes)	160
XI.	Timing Data: Program Restart/Relocation	164

LIST OF FIGURES

Figure		Page
1.	Intel 432	13
2.	STAR	14
3.	FTMP	18
4.	PLURIBUS	26
5.	SIFT	32
6.	AT&T 3B20 Duplex Computer	35
7.	Cm* Hardware Structure	61
8.	Computer Module (cm) Organization	62
9.	PLITS Distributed Jobs	70
10.	Planar-2 Network Topology	98
11.	AOSP APU/NCU Planar-4 Connection Scheme	102
12.	FTM System Overview	108
13.	FTM System Communications	110
14.	FTM Main Loop	130
15.	Small Sample FTM Session	142
16.	Distributed System for Performance Tests	150
17.	Summary of FTM Call Times	165
18.	Summary of Message Transmission Times	166
19.	Simple Network Topology	167
20.	System Reliability on 5 of 16 Nodes	168

CHAPTER I

INTRODUCTION

1.1 Motivation

Historically, hardware fault tolerant computer systems have been designed for specific applications and offered this fault tolerance through massive hardware replication at costs far exceeding general purpose systems of equivalent computing power. The plunging cost of computer hardware allows for economic inclusion of fault tolerance into general purpose hardware systems. Since mean time between failure (MTBF) has always been a major hardware evaluation factor, it is understandable that the level of fault tolerance included in even general purpose hardware has been dramatically increasing.

The evolution of software fault tolerance has not been as dramatic. There have been software based special purpose fault tolerant systems. These generally rely on software to make switching decisions among replicated hardware modules. A more recent trend is to take a system with hardware components which are replicated for performance rather than fault tolerance, such as multiprocessors and networks, and to layer upon this hardware a software support system which allows application programs to access fault tolerance services.

This research will define a software environment in which applications will be able to access services to aid in recovery and continued operation in the face of software and hardware failures. The emphasis will be

---

The *Communications of the Association for Computing Machinery* is used as a pattern for format and style.

on layering these services on top of a general purpose operating system running on a general purpose system. The aim is to allow applications to make full use of all normally available system resources and in addition to have these fault tolerance services available.

The fault tolerant environment defined by this research will be generic in the sense that it will be independent of any specific hardware configuration or operating system. It will be implemented on a general purpose computer system neither designed specifically for, nor dedicated to, the generated fault tolerant environment.

1.2 Approach

First, a literature survey provides past and current approaches to fault tolerant environments. Next, initial decisions as to the general approach must be made and justified. Minimal hardware requirements are defined and protocols are developed to implement the software environment. An existing local available system is chosen upon which a prototype is then implemented.

The prototype system is then used as a platform to study the implemented fault tolerant environment. A variety of tests are run under the environment both to demonstrate the validity of the approach and the generate performance statistics. The performance versus fault tolerance (and convenience) trade-offs are considered and discussed.

1.3 Overview

Chapter II presents an extensive literature survey. In Chapter III, the chosen approach is described, along with a discussion of the prototype environment. The complete application user instructions, along with the

code, is given in Appendix A. Chapter IV describes the user interface and protocols for the fault tolerant environment. Chapter V presents the results of validation and measurement tests run against the implemented environment. Finally, Chapter VI presents conclusions and suggested areas of further investigation and research.

CHAPTER II

REVIEW OF LITERATURE

2.1 Introduction

The volume of literature that is available and relates to fault tolerance is so large that necessary sub category classifications and detailed fault tolerance requirements are essential to restrict the literature survey. Traditional computer systems typically demonstrate software, hardware, and system failures that cannot totally be eliminated by after the fact fault tolerant approaches. Rather, a reasonable choice of fault tolerance features based on environment specific fault tolerance requirements, would provide reliable computing. Various classifications of fault tolerance attributes help in choosing attributes most likely to be beneficial in a specific critical application.

2.1.1 Classifications

Rennels [68] provides an excellent overview to the history of fault tolerant computing from the early 1960's. Besides providing high level descriptions of numerous landmark fault tolerant systems, he classifies several aspects of fault tolerance into clearly defined groups.

The most general classification of fault tolerance is by requirement. Although the need for fault tolerance may be equally critical across various computing environments, the specific recovery requirements may be totally unrelated. Rennels divides requirements into three categories: fault coverage, computational integrity, and time between maintenance. Fault coverage is the determination of which types of faults have the highest detection and recovery need (i.e.: which faults are "covered"). So,

the system fault coverage against a specific fault is the probability that an instance of this fault type will be detected and recovery action taken. Fault coverage becomes most critical in such applications as nuclear reactor safety systems or manned space vehicles, where incorrect outputs can result in death or mass destruction. Computational integrity is the ability to ensure that computed data is correct. This is most important in such applications as financial systems. Sensor based systems can usually recover from an absent or incorrect data point because the data is constantly being updated and historical data is of lower importance. However, an item of financial data is generally folded into a running total so its loss or corruption causes problems which are not transient in nature. Time between maintenance is self-explanatory. Its importance lies in the determination of how critical a period of down time would be. In the case of non-repairable systems (e.g., remote systems such as space vehicles) this category becomes the most important.

Rennels further subdivides fault causes into three classes: physical faults, external errors caused by the environment, and design faults. The most difficult to provide fault tolerance for are design faults. Note that all software faults fit into this category. Validation of designs is very difficult for all but the most basic of hardware modules and prohibitive for large software systems. One approach to design fault tolerance is to compare the output of several completely independently designed and built modules.

Rennels again uses three categories to classify hardware redundancy: triplication with voting, duplication with comparison, and standby replacement. Hybrids of these categories also exist. Systems using each of these approaches will be briefly described.

2.1.2 Survey Articles

Other survey articles including that by Avizienis [8] describe the history of fault tolerant computing at the Jet Propulsion Laboratory and at UCLA. These systems are generally designed for avionics and space applications and include such early fault tolerant systems as the STAR (self-testing and repairing) and UDS (unified data system). Serlin [73] describes fault tolerant systems, designed for commercial applications, such as the Tandem Non-Stop and Stratus. These systems emphasize computational integrity.

2.1.3 Mathematical Classifications

A number of articles classify fault tolerance systems mathematically. These systems can be classified as variations of queueing models [10, 29]. Availability can be described using Markov state models [30] and reliability analysis models [62] can be used to aid in configuration decisions. Siewiorek [78] reprints a variety of detailed descriptions of early fault tolerant systems.

2.2 Hardware Fault Tolerance

This section presents an overview of hardware approaches to fault tolerance. It must be noted that although these techniques can and have been used for software implemented fault tolerance, they often have an increased negative effect on system performance.

2.2.1 System Level Considerations

The difficulty of embedding fault tolerance into large, complex computer systems [38, 47] shows the need to design fault tolerance into the basic architecture [77] of the system, rather than to add these features as

afterthoughts. Architectural considerations include: levels and arrangements of component replication, component reliability prediction, physical layout (to isolate sensitive components from mechanical, chemical, and electrostatic discharge hazards), diagnostics, software restrictions for reliable programs, etc.

Another consideration, at the system level, is the determination of optimal retry policies [48]. A dynamic retry policy will base the current retry actions on knowledge about the current fault. Hardware data structures [58] can be designed to specifically cover faults using spare hardware (such as extra memory cells).

The architecture should attempt to minimize correlated failures [33] among system components. Markov modelling of system reliability assumes single state transitions. That is, a second failure does not occur until action has been completed on the first failure. The fact that allowing simultaneous failures complicates the mathematical model of a system would by itself be a poor reason to expend cost and effort in their avoidance. But, simultaneous failures also complicate actual required recovery paths far beyond those paths required to recover from consecutive, non-simultaneous faults.

A system with duplicated hardware can often be modelled as a series-parallel system. While an attempt at a fully accurate model of any reasonable system would quickly become enormously complex, empirical studies [56] have shown that many simplifying assumptions can be shown to have little effect under reasonable probability distributions for failure rates and repair delays. Availability characteristics can then be generated from this series-parallel model.

One approach that is used in the development of a fault secure large system is to concentrate on the interfaces between subsystems. Studies [57] have shown that the design of subsystem interfaces can incorporate the concepts used in the well studied totally self-checking (TSC) network model, namely all outputs follow an encoding scheme. Since an error output would be unlikely to be a legal encoding, a checker should be able to detect the error before it is propagated. Large systems are difficult to model, so the injection of errors into actual systems [17] can be used to aid in measuring the ability of the system to isolate and compensate for such errors.

As the reliability of individual components increases, measuring data from implemented systems becomes less feasible, and modeling [36] increases. Models, especially Markov and semi-Markov models, are needed to produce mathematical predictions on reliability levels.

2.2.2 Approaches Using Additional Hardware

Hardware redundancy can be described from several aspects. All of the techniques are from the hardware viewpoint, although equally applicable to software implementation, either alone or in concert with the hardware. As always, efficiency, cost, and requirements for fault coverage govern the optimal techniques and approaches to specific cases.

2.2.2.1 Dual Components

A dual system is an example of *replication*. This technique uses duplicate copies of a hardware component, such as a CPU, a memory module, a disk, etc., where there is a primary module and either a shadow or a standby. In the case of a shadow system, the secondary module immediately and transparently switches to become the primary mode to

prevent system failure. The failure can either be indicated by the primary, or, in the case of inconsistent outputs processing can be halted to allow each module to run a self-check. Replication can be generalized to include, for example, four modules connected into two sets of two modules, each set acting as above, with the adddition of a comparator. Standby systems have obvious disadvantages in the delays needed to implement checkpoint type restarts. Thus, these restarts may be infeasible for certain applications. Their main use is in unattended systems (since wearout is minimized) and systems with limited available power. Replacement policies in replicated systems have been exhaustively studied [11, 52] and depend on the desired optimal fault coverage.

2.2.2.2 Triple Modular Redundancy

Triple modular redundancy (TMR) uses three copies of a module (actually any odd number of modules may be used) with the outputs gated to a voter circuit. The voter circuit is assumed to be *much* simpler than the triplicated modules and therefore inherently more reliable. Comparisons between the replication and the TMR approaches [87] show advantages for each. The TMR approach, although guaranteed to detect and mask any error in a single module (even if the module produces a worst-case output) has a higher hardware duplication. After a disagreement has occurred and the voter has logically disconnected one of the triplicated modules, the problem becomes one of deciding which of the remaining two modules to accept for any future disagreement. The addition of replication techniques to the module circuits (like shadowing and self-checking abilities) would allow further fault tolerance after the first failure. This suggests a hybrid replication-redundancy system.

The TMR technique can be generalized into majority voting in the case of large numbers of voters. For example, in the case of a large distributed network of processors agreement on certain issues, such as the current time or the status of a possibly aberrant system, can be voted on by the systems and the results broadcast. Majority agreement in the face of a large voting population, possibly including aberrant voters, can be shown [21] to reliably reach consensus in a short time even in cases of surprisingly unreliable individual systems.

2.2.2.3 Watchdog Processors

Another hardware approach, that of the watchdog system [51], uses a separate processor to monitor various messages between modules. This is a generalization of the watchdog timer concept. Various techniques have been studied to implement watchdog processors, which can be used to monitor any combination of: memory access behavior, control flow, control signals, and reasonableness of results.

The watchdog processor itself must be reliable, which implies that it is simpler than the system being checked. In addition, the ability of the watchdog processor to detect abnormal conditions is based on the watchdog scenario and understanding of normal system behavior. Simple faults can be detected by such factors as timeouts, with the cooperation of the modules in the system. Reasonableness checks require an understanding of the meaning of the exchanged data between modules. Signature data attached to each data stream allows the watchdog processor to monitor expected outputs from modules.

2.2.2.4 Design Diversity

Avizienis and Kelly [7] present a good survey of a technique called

design diversity. The idea is to duplicate the functionality of a module, but not the module itself. That is, several different teams independently design and construct a module. Replication and/or redundancy techniques are then employed with these independently produced modules. This technique will overcome a design fault, where the design of a module causes it to incorrectly handle some particular input data. This assumes that each of the modules was designed from the same system specification.

> *The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods.*
>
> Dionysius Lardner
> "Babbage's Calculating Engine"
> *Edinburgh Review*
> July, 1834

2.2.3 Hardware Fault Tolerant Systems

Several examples of hardware fault tolerant systems will be presented next. A *hardware* fault tolerant system is one in which the hardware, rather than the software, is primarily responsible both for fault detection and for directing the recovery efforts. The Intel 432 system assigns almost total responsibility for fault tolerance to the hardware. However, most hardware based fault tolerant systems detect faults at the hardware level, but may involve some direction from software for recovery procedures. Often the operating system attempts to isolate other software from the hardware fault tolerance features [94] by directing the testing, reconfiguration, or graceful degradation of the system in response to detected faults. The software fault tolerance routines are therefore designed primarily to exercise the hardware fault tolerance features.

design diversity. The idea is to duplicate the functionality of a module, but not the module itself. That is, several different teams independently design and construct a module. Replication and/or redundancy techniques are then employed with these independently produced modules. This technique will overcome a design fault, where the design of a module causes it to incorrectly handle some particular input data. This assumes that each of the modules was designed from the same system specification.

> *The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods.*
>
> Dionysius Lardner
> "Babbage's Calculating Engine"
> *Edinburgh Review*
> July, 1834

2.2.3 Hardware Fault Tolerant Systems

Several examples of hardware fault tolerant systems will be presented next. A *hardware* fault tolerant system is one in which the hardware, rather than the software, is primarily responsible both for fault detection and for directing the recovery efforts. The Intel 432 system assigns almost total responsibility for fault tolerance to the hardware. However, most hardware based fault tolerant systems detect faults at the hardware level, but may involve some direction from software for recovery procedures. Often the operating system attempts to isolate other software from the hardware fault tolerance features [94] by directing the testing, reconfiguration, or graceful degradation of the system in response to detected faults. The software fault tolerance routines are therefore designed primarily to exercise the hardware fault tolerance features.

2.2.3.1 Chip Level Considerations

Possibly the earliest approach to fault tolerance in computer systems involved error detecting codes in I/O transfers from/to peripheral devices. A simple lateral parity bit per byte could detect any single bit error. The addition of a longitudinal parity frame allowed the detection of any double bit error in a block of data. Typically, this information would be made available to the software which would be responsible for appropriate reporting or recovery. The next step was to apply this concept to memory data transfers [14, 71]. Simple parity checking quickly gave way to Hamming *error correcting codes* (ECC), which can correct single bit errors. Usually, each individual byte is Hamming protected. The addition of a parity bit allows for the detection of all two bit errors within a byte. This code (Hamming with a parity bit) can correct any single bit error and detect any double bit error. It is known by its acronym SEC-DED, which stands for single error correction – double error detection. This fault handling is the total responsibility of the hardware, except possibly for a logging procedure to allow the software to migrate data from a repetitive memory bit failure.

Advances in VLSI technology migrated increasing levels of fault tolerance onto each individual chip. Mathematical modeling methods can be used for VLSI device reliability predictions [27] covering both structural and performance defects.

2.2.3.2 Intel 432

The Intel 432 (see Figure 1) is a system based on fault tolerance at the chip level [40, 96]. Besides self-checking and correcting circuits built into each chip, the architecture allows chip modules to check and complement each other at the hardware level with minimal software

intervention. For example, if two CPU modules are present, the second may act as an independent CPU or it may shadow the primary, using its comparator circuit to signal any detected inconsistency. This is also implemented with memory modules. Memory modules contain a standby bit chip (each 32 bit word has 40 individual bits: the 32 data bits, 7 ECC bits, and the standby bit) so that a hard memory error can be corrected by the memory control unit, by simply assigning the standby bit chip to replace the failed bit chip.

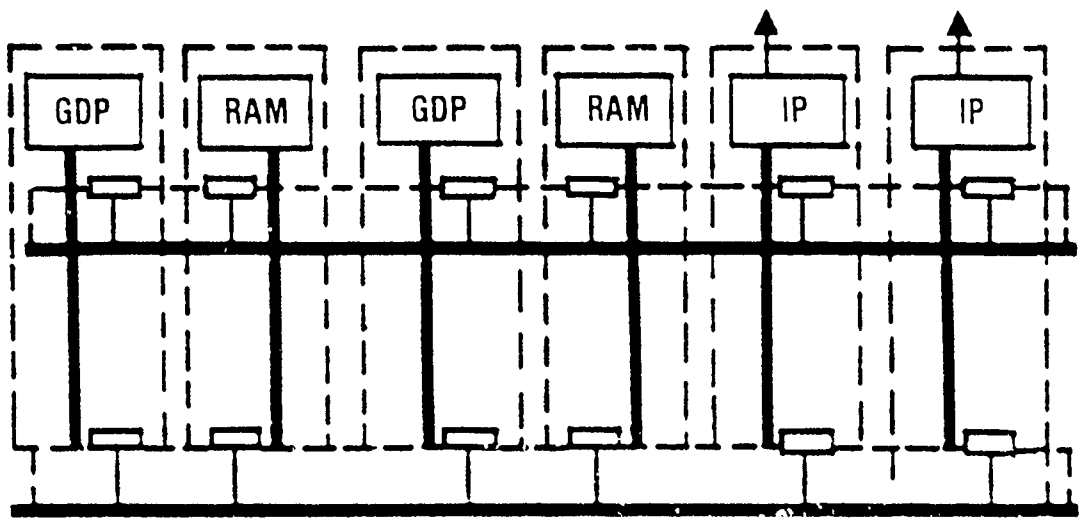

Figure 1. Intel 432 [40]

The architecture supports a fully object-oriented programming approach, with an operating system that can decide on the level of fault tolerance, and if necessary assign duplicate modules (such as CPUs or memory modules) to be either shadows or independent components. The hardware can then manipulate the software objects as needed. The application software is totally ignorant of the current configuration and the operating system need only define the current configuration (along with setting constants such as retry limits) leaving the fault detection and

recovery to be a strictly hardware level operation.

While the Intel 432 is significant as an attempt to implement fault tolerance almost totally at the hardware level, it should be noted that it was not a commercial success. The minimal operating system and the almost non-existent application software probably did not impress potential users any more than the fact that the only available language compiler was ADA.

2.2.3.3 Self-Testing and Repairing (STAR) Computer

The STAR computer (see Figure 2), designed at the Jet Propulsion Laboratory (JPL), had the objective of adapting the system as the central computer of a space exploration craft [6]. The system design is based on the requirements to perform reliably for a long period of time (several years) in a remote location under severe space and power restrictions.

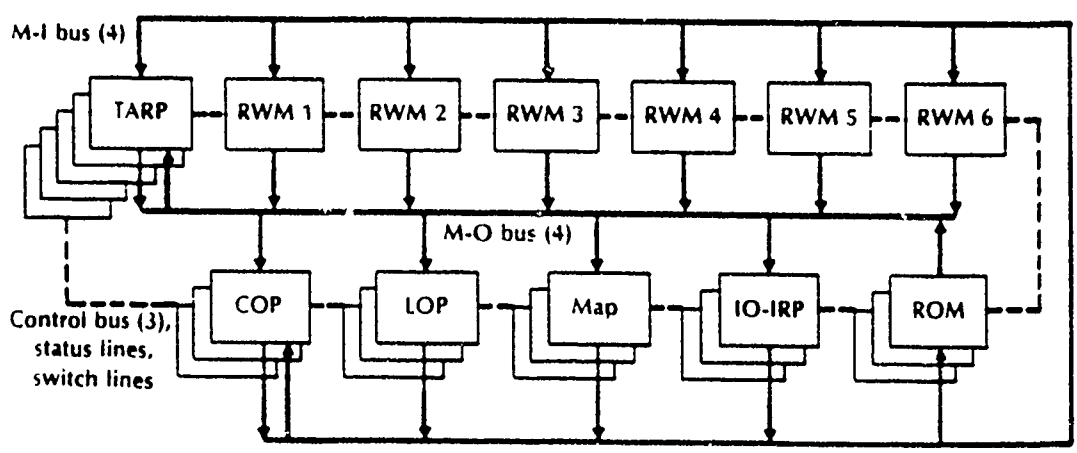

Figure 2. STAR [6]

Although static redundancy (such as triple modular redundancy) provides immediate fault masking, the expense of higher power usage and wear-out of all components motivated dynamic (standby) redundancy.

Dynamic redundancy meets the power restrictions (only active processing units are powered) and the inability to physically replace failed components increases the importance of the wear-out factor. Keeping spares unpowered delays these wear-out effects.

Other advantages of dynamic redundancy include the ability to easily adjust the number of spares, the survival of the system until all copies of a component have failed, the avoidance of synchronization complications, and the isolation of unpowered spares from catastrophic environment faults.

One potential problem with standby redundancy is that with only one powered unit of a particular module type it becomes critical that faults be detected either through self-checking or through the use of a watchdog. STAR employs both of these.

If a data item or instruction is altered during storage, transmission, or processing, the result is a *word error*. Data duplication provides the greatest fault coverage for word errors, but at the unacceptable cost of having double powered memory and processing units. STAR therefore uses an arithmetic error-detecting code. Each 32 bit word has 28 data bits and 4 check bits. All instructions and data items are 28 bits and all arithmetic operations have 28 bit results encoded into 32 bit words.

A *control error* by contrast results from the incorrect execution of an instruction. STAR employs a 4-wire status line for each unit to send status messages. The system works on the concept of a ten unit (ten clock ticks) cycle of operation, simplifying expected status message synchronization.

Because the execution of each instruction takes exactly ten steps and synchronously creates status messages, the concept of a watchdog can be readily employed. Called a Test and Repair Processor (TARP), this unit is the heart of the STAR hardware fault tolerance. The TARP receives the status messages from the other units (control processor, read/write and read only memory modules, arithmetic and logic unit, I/O processor, etc.) and uses them to monitor the system operation. It is the TARP that detects an error and determines appropriate corrective action, such as to power down a component and power up one of the standbys. While the TARP is controlling the system, the system is said to be in *recovery mode*. When recovery is complete and the TARP returns to a monitoring function, the system resumes *standard mode*.

In a TMR system, the voter, a critical point of failure, must be much more reliable than its feeder modules. The TARP plays a similar role in the STAR. However, the fact that it tests for and repairs faults in all other modules means that it cannot be designed to be much simpler and more reliable than the modules that it is monitoring. So, the TARP cannot be treated in the same "one unit up" manner as other components. In fact, there are three powered and two standby TARP modules, with a voter testing the results.

Another capability besides self-checking of a standby redundancy system must be the ability to roll back programs. While in some cases the TARP may be able to have the processor retry an instruction, in many cases the repair is much more involved. A memory failure, perhaps an entire memory module, requires a program restart. All programs running on STAR are therefore expected to take advantage of the checkpoint facilities, so that if needed the operating system can respond to a TARP

interrupt and checkpoint restart the affected process.

Although the STAR was designed for fault tolerance from a hardware approach, the system software provides a user interface (especially important because of the requirement for programs to checkpoint themselves) along with high level decisions (acting on information from the TARP) and restart and cold start procedures. The system software designed subsequent to the hardware has as its main purpose the support of the hardware fault tolerance features.

2.2.3.4 Fault Tolerant Multiprocessor (FTMP)

Like STAR, the FTMP system (see Figure 3) was designed as a critical avionics controller with no external repair capability during its mission [35]. However, the similarity of application ends here. The FTMP is intended for use as a normal (i.e., earthbound) aircraft support system. This relaxes some of the restraints placed on the STAR design, while increasing other constraints.

Size, weight, and power restrictions are among the most critical constraints on systems designed for space travel. While not totally absent, these particular limitations pale in the comparitively less restrictive aircraft environment. Unattended operation (including lack of repair during operational use) is common to both environments. However, an aircraft mission is on the order of 10 hours, after which repairs can be made, exhaustive off line testing can be performed, and components can periodically be replaced.

Availability is a factor which is more critical in the aircraft environment than in space applications, especially unmanned space applications. A highly maneuverable aircraft, such as a fighter plane, is often in

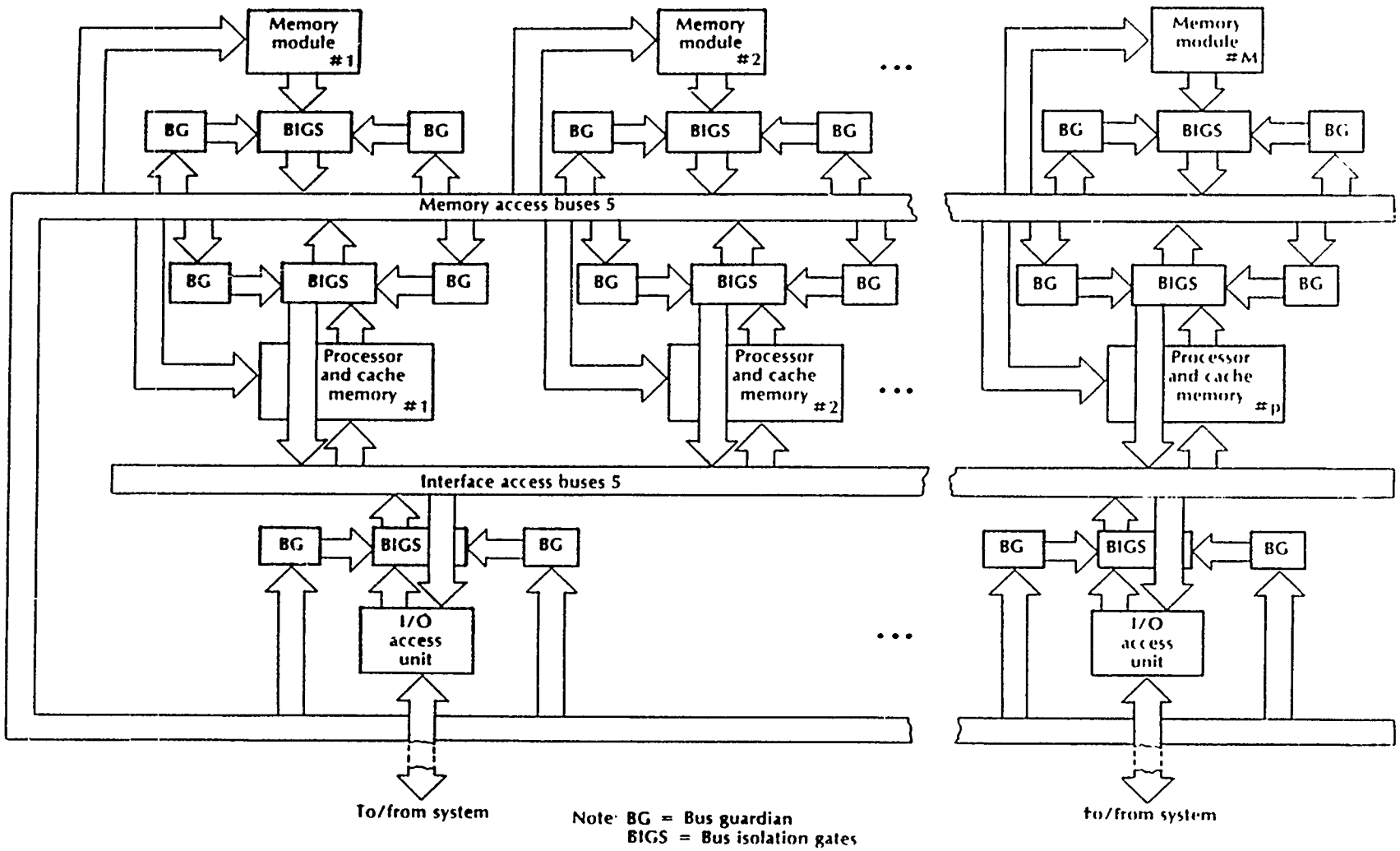

Figure 3. FTMP [35]

positions where loss of control for a fraction of a second could be critical.

Given the divergent nature of the environment requirements for space applications and aircraft applications, it is natural that the design for the STAR and FTMP systems also diverge. The primary restrictions of volume, weight, and power for the STAR have been supplanted by the requirement for high availability of the FTMP.

FTMP is based on the TMR-hybrid approach to fault tolerance. That is, components are clustered into *triads*, each consisting of three identical modules with a voter. In addition, spares are available to replace any member of any triad that has been declared in error. Assuming reliability in the voters, fault coverage is complete against any single error. A double error may possibly be catastrophic (such as two modules failing within one triad), but only if the second occurs within the time needed to assign a spare.

Because errors are masked by the TMR voting within each triad, software restrictions are much reduced from the STAR approach. Checkpointing and rollback are not required, although some reasonable software support (such as system restart) should be allowed for recovery from catastrophic failures. This greatly increases availability over the STAR approach, where any non-recoverable error results in a program rollback. This FTMP program rollback is relatively time consuming when compared to the STAR hardware controlled reconfiguration with program continuity.

In addition, FTMP is a true multiprocessor. That is, there are multiple CPU triads and the operating system assigns processes to processors according to priority and availability. So, even if spares become exhausted the system can operate in a degraded mode with fewer processor triads. A

similar concept applies to memory triads.

The physical organization of the FTMP is very complex. Each triad can be composed of any three physical instances of the appropriate module type. All components, including processors, memory modules, and busses, can be arranged in triads. The complex circuitry needed to control these triad configurations is in addition to that needed to control the usual tightly coupled multiprocessor complications such as cache coherence. To control the current hardware configuration, each processor, memory, and I/O control unit, includes a sub-module called a *bus guardian*. Each guardian has the responsibility to control the status of its own module; whether or not to apply power to the module; to select the correct active attached bus; and perform self-test analysis (only if the module is not currently part of an active triad).

Guardians can be biased toward the "safe" direction for each selection, and thus increase reliability even more. For example, the "default" action can be power off and a disconnect from all busses. This can be enhanced further by supplying multiple guardians for each module, with required guardian agreement before changing the default action.

The guardian, the point of isolation for a module (the guardian selects the active bus connection), becomes the critical point to isolate failures. Guardians are therefore designed to be as independent as possible. Each receives power and timing references independently of all other guardians. The bus isolation gates are critical in the protection of the physically attached busses. They, as well as the guardian circuits which enable them, must be independent. Each guardian is physically isolated from all other guardians and all modules.

Reliable clocking is critical in any TMR system, since the voters must receive synchronous inputs. Several approaches have been tried. The first was a redundant clocking arrangement based on a majority logic algorithm. Another uses four phase-lock oscillators, with clock receiver circuits in each user module. In either case, there are 5 clock busses with 4 active at any one time. Each guardian is responsible for controlling the clock receiver circuit in its module.

Because the FTMP is a multiprocessor, on-line testing can occur while critical processes continue to run. Software routines can cause active processor triads to enter a test mode while other active processor triads execute critical processes. While in a test mode, a triad can test its voter circuit by purposely providing distinct outputs from the voted modules. Although bus triads can be tested in a similar fashion, memory triads cannot be tested on-line.

It should be noted that engineering prototypes of the FTMP have been constructed that employ some, but not all, of the above redundancy principles. For example, one such prototype from the Charles Stark Draper Laboratory employs the concept of a line replaceable unit (LRU). Each LRU consists of one processor with cache, one memory module, one I/O port, one clock generator, and related control and peripheral circuitry. The prototype consists of 10 LRUs. This allows for three processor triads, with the tenth processor as a spare. The three memory triads (again with one spare) allow for 48K (3 * 16K) of shared memory. Each LRU contains a common power supply for its contained modules. The components within each LRU are logically independent, but not physically. The reduced reliability, from the physical dependance, is accepted because of the resulting simplified physical implementation.

2.3 Software Fault Tolerance

Software fault tolerance refers to the masking of or recovery from faults in software. The use of software to aid in the tolerance of hardware faults is a distinct subject. As has been previously noted, all software faults are design faults. Many techniques of hardware fault tolerance are therefore inappropriate to software approaches. For example, wear-out is not a factor. Replicated copies of a software package are useless against a software fault. On the other hand, approaches to tolerance of hardware design faults can be easily transformed into equivalent approaches to software fault tolerance.

2.3.1 Software Approaches

The best way to provide tolerance from design faults in hardware is to avoid them. That is, design the hardware correctly to begin with. It is the increasing complexity of the hardware that makes this goal difficult to reach. Despite the fact that a given hardware design is likely to result in the production of a large number of physical units of that design, it is still prohibitive in cost (dollars and time) to assure design correctness beyond a certain point. A typical software package can be many orders of magnitudes more complex, and correspondingly more difficult to prove correct. For this reason, the historical approaches [75] to software reliability have assumed bugs in the software and described software reliability in terms of probabilities, such as: *Software reliability is the probability that a given software system operates for some time period without software error, on the machine for which it was designed, given that it is used within design limits.*

The *robustness* of a program is defined as its ability to perform according to its design over the range of legal input. Any input will be

considered legal, with erroneous input resulting in a detected exception. Preventing robustness failures in a software system is the goal of software fault tolerance. Everhart [24] divides robustness failures into 24 categories, along with approaches to handling each failure category.

Software fault tolerance initially involved hardware features [41, 61] to isolate software modules by restricting hardware privileges. Memory keys (and later virtual address translation tables) were used to restrict which memory locations could be accessed by various software tasks, and hierarchical protection levels restrict tasks from performing sensitive processor operations. The idea is to restrict critical operations to well designed and tested software (such as the supplied operating system) while less trusted programs would have to use defined interfaces requesting critical actions to be performed on their behalf.

As Lardner's quote from "Babbage's Calculating Engine" stated, design diversity to design fault tolerance is appropriate both in hardware and software. It is in general much easier to have several independently designed and coded software modules than to have several independently designed and manufactured hardware components. Furthermore, these software modules can be run independently on one processor or in parallel on several different systems, with subsequent voting and comparisons.

Randell [64] formalizes the concept of software design diversity through a mechanism called a *recovery block*. This is a conventional block of code, requiring only that explicit points of entry and exit be defined, which is supplemented by several constructs. An *acceptance test* provides a means of error detection. And, one or more *alternative blocks* are included. Any time the recovery block is executed, the acceptance test is performed prior to exit. A failure results in an invocation of the first

alternative block. The acceptance test is repeated before this block is exited. Another failure results in an invocation of the second alternative block. This sequence continues until the final alternative block defined for this recovery block has been invoked. An acceptance test failure at this point is reported as an unrecoverable error from the recovery block.

Data diversity [3], a design diversity derivative, decomposes the input data such that each component is independently used by the software system, recombining the results and comparing this result with the original composite input. The feasibility of this approach depends on the function of the software module. For example, consider the following calculation:

$$y = sin(x)$$

Since:

$$sin(a+b) = sin(a)cos(b) + cos(a)sin(b)$$

$$cos(a) = sin(\frac{\pi}{2} - a)$$

select any $\{a,b \mid x = a+b\}$ and calculate:

$$y_2 = sin(a)sin(\frac{\pi}{2} - b) + sin(\frac{\pi}{2} - a)sin(b,$$

It is now possible to compare calculated values for y and y_2. Since the sin routine had been tested over a wide range of input values and the probability of a design fault for any particular input is low ($p << 1$), then the probability of having offsetting errors is yet even lower (on the order of p^2).

2.3.2 Software Fault Tolerant Systems

In this section several systems designed for software based fault tolerance implementation will be described. As in previous discussions on hardware fault tolerant systems, the hardware and software are designed to complement each other and both contain features specifically meant to

provide fault tolerance. The difference is in emphasis only; the following systems place most of the responsibility to detect faults, and especially to recover from them, in the software.

2.3.2.1 PLURIBUS

The PLURIBUS multiprocessor system (see Figure 4) was designed specifically as a second generation replacement for the ARPANET interface message processor (IMP) [42, 59]. The purpose of the IMPs, acting as a message routing system, provide the network backbone. Once a message was received from a sender, the IMP became responsible for routing it to the next IMP in the chain until it reached the IMP from which it could be delivered to the receiving node. Fault tolerance for the IMP system was basically limited to the presence of multiple paths between IMPs. Of course, any messages held by an IMP were lost during IMP failure, and, if the IMP connected to the sender or receiver failed, then the path was considered disconnected pending IMP repair.

The overriding constraint on PLURIBUS, being a message routing system, was availability. Thus, a reasonably infrequent loss of data could be tolerated, since upper level protocols at the sender and receiver are responsible for assuring reliable message delivery. The systems would be located at large sites where MTTR could be minimized, operating in a degraded mode, while awaiting repair (rather than being down completely).

PLURIBUS, a single purpose computer, emphasized duplicated hardware with coordination by software where the system software and application software were designed as a unit. And, since the system application emphasized I/O (message receives and sends), free CPU cycles were

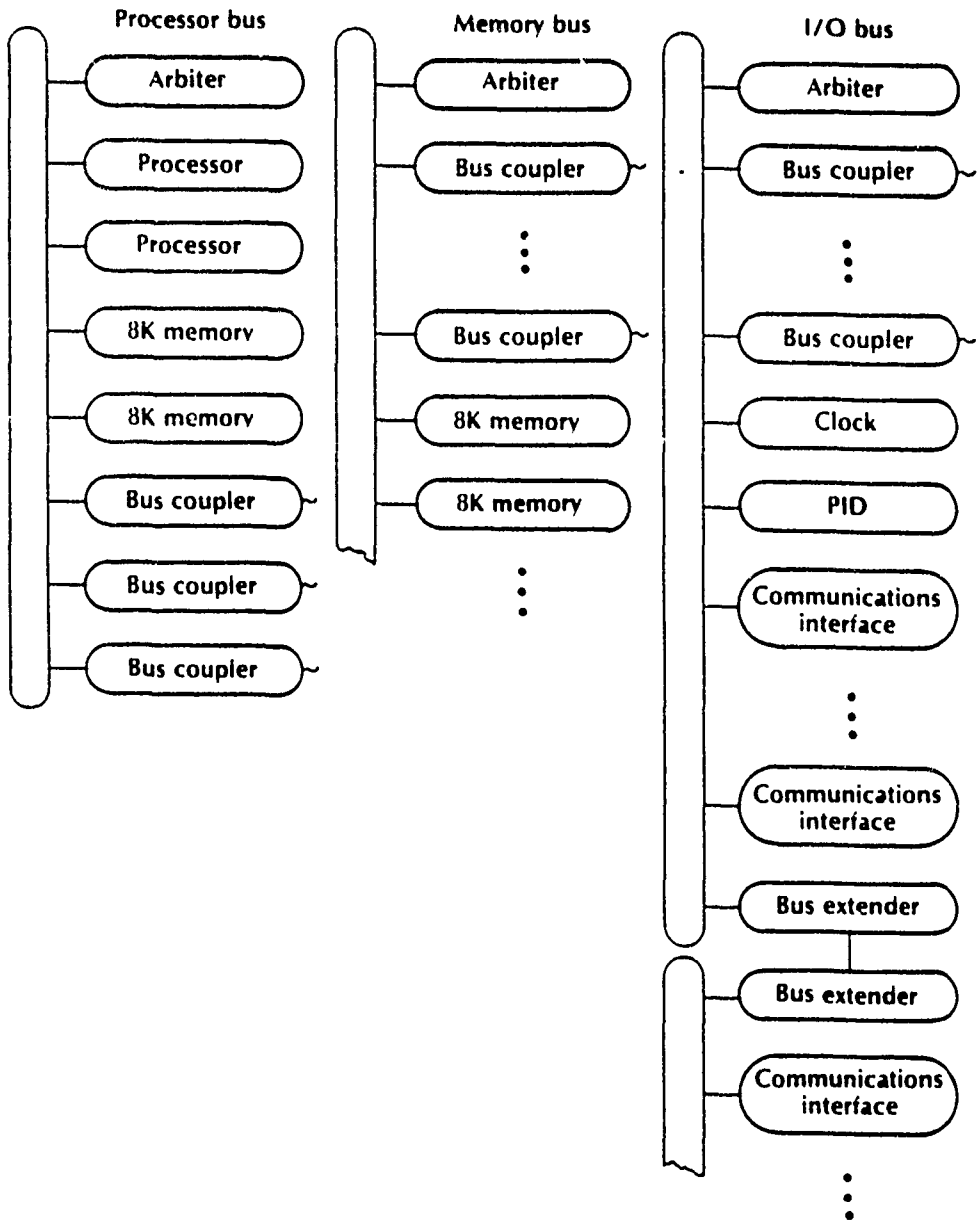

Figure 4. PLURIBUS [42]

available to handle fault tolerance overhead.

Each PLURIBUS system contains multiple identical processors. In each PLURIBUS system the ideal number of processors is the number necessary to handle the expected load, plus one. Thus, the failure of any

single processor results only in the loss of excess throughput capacity. These processors are tightly coupled, sharing memory and all other system resources. System components are independent in terms of environmental factors such as power and cooling. The interconnection between modules is a distributed form of a crosspoint switch. A maximum of eight processors and eight memory modules are allowed.

All modules are activated either at power up or reset time. Each processor loads its local memory from an external source, and then attempts to communicate with the other processors. After system consensus is reached, the system software loads applications into main memory and creates data structures. Each of the identical processors then becomes eligible to run any task. All interprocessor communication is performed in a section of main memory called the *communication page*. Hardware fault tolerance is limited to parity checking between modules and timer interrupts to indicate lack of expected responses.

The coordination of processors is accomplished through the *logical system configuration*, a data structure in the communication page. All changes to the logical system configuration must result from a consensus of the processors. The approach is not to immediately detect a fault, but rather for each processor to periodically check to see if its view of the current system matches the current logical system configuration. If not, a request (through the communication page) is made to force a consensus on the potential discrepancy. Any processor not agreeing with the consensus must terminate its place in the system. Errant processors are designed to quit rather than continue to destructively interfere with the remaining operational part of the PLURIBUS system, and thus Byzantine behavior need not be logged.

The software of each processor, utilizing a sequence of ten initialization *stages*, performs self diagnosis and system checking before the processor is allowed to join the active system. Each successive stage builds on the self-tests performed at the earlier stages. During processing, the checks associated with each of the stages are periodically retried. Any discrepency results in complete reinitialization of the system before the final stage (the application system) may be entered.

Coordination among processors is by way of a common communication page in memory. The first step is for the processors to agree on a page. Since memory is not replicated, the loss of a memory module results in the loss of a specific part of the address space. Thus, processors may be unable to access certain memory pages. The agreement on a communication page is a critical operation because processors, expected to coordinate their own non-interference with other processors, would immediately encounter critical interference in the presence of duplicate communication pages.

Each page in memory contains a small area reserved for communications data, including a pointer to the communications page. The idea is that a processor expects the communication page to be the lowest accessible numbered page. If the pointer in that page points to itself, it identifies the communication page. If the pointer is to a higher numbered page, then the processor defines its low page as the communication page and fixes all the pointers in the pages it can access. If the pointer is to a lower page, then the processor must determine why it cannot see that lower numbered page. It may be that a hardware failure has made that page inaccessible (but accessible to other processors) which means that the processor must shut itself down for repair. Or, it may be that the memory

module is bad, the page is not accessible to any processor, and this is simply the first processor to notice this condition. In this case the processor should increment all pointers to point to the new (higher) low numbered page. Since there is currently no coordination between processors, a mechanism must be used to determine whether the processor shuts down or defines the higher numbered communication page. The mechanism is as follows: all processors with a successfully defined communication page periodically zero a timer in the communication area of all pages. So, the processor that is deciding whether to define a higher numbered communication page must check the timer on that page. If the timer does get zeroed within a given period of time, some other processor is successfully using both that page and the indicated lower numbered communication page, which means our processor must stop itself rather than attempt to change the communication page.

This complicated procedure to select a communication page has high continuing overhead, and furthermore allows a worst case scenario where the processors can be partitioned into multiple classes, each with their own communication page. This scenario in which all of memory is partitioned so that there is no memory interference between processor groups, does result in system failure because of I/O bus interference. This is an example of Byzantine failure which the PLURIBUS design is not able to handle.

Two general techniques are used to ensure data integrity in the PLURIBUS system. The first, and standard, approach is redundant information. Significant system control data structures are duplicated and periodically compared. Detected differences result in attempts to correct the situation, with early detection minimizing the spread of the fault.

The second general technique is the use of watchdog timers. Timer resets are used to aid in the determination of a common communication page, and also to manage data structures. In a message handling system, such as PLURIBUS, buffer pool management is critical. This is especially true since the loss of a message is not considered a critical failure as long as the system continues operation. Problems with the buffer pool would represent significant degradation. Flags are set on message buffers to indicate their appearance on the free list. Any buffer which has not appeared on the free list for more than two minutes is considered "stuck" and is forced onto the free list despite the possible loss of any message data.

PLURIBUS seems to show that software implemented fault tolerance compares poorly to the previous examples of hardware based fault tolerance. PLURIBUS has complicated software coordination mechanisms, devotes a communication area on every page (along with periodic associated timer updates on every page) simply to assure further processor coordination, counts on each processor to check itself and refrain from Byzantine behavior, expects each processor to periodically retry its multiple stage self-checks, etc. Data integrity is not even a design goal; communications software on other computer systems have the responsibility to check for missing or garbled messages.

One very significant advantage of the PLURIBUS approach is cost. Most of its hardware is standard off-the-shelf. Only the distributed crosspoint switch is not generic. Additional modules (processors, memory) may be added with little additional hardware complications. Fault tolerance features may be tested and evaluated in software at minimal cost. Revised software version can be distributed much more easily than a set

of hardware field changes.

Another advantage is error reporting. While hardware fault detection and correction usually results in some logging information, software error handling provides detailed records. This more complete error reporting can be used to better understand the nature and frequency of various faults that occur during operation.

2.3.2.2 Software Implemented Fault Tolerance (SIFT)

SIFT (see Figure 5), closely matching the FTMP hardware based fault tolerance system, is intended for use as an avionics control system with requirements for ultra-high availability over a relatively short period of time (on the order of ten hours) [93]. Unlike FTMP, however, SIFT places the fault tolerance responsibilities mainly at the software level.

Similar to the PLURIBUS software directed fault tolerance system, SIFT can depend on mostly off-the-shelf hardware. SIFT, because of different design and operational requirements, can tolerate neither downtime nor repairs during system usage. In fact, the system restart and reconfiguration allowed in PLURIBUS is not acceptible in SIFT. The fault tolerance emphasis in SIFT is therefore software level TMR.

SIFT contains up to eight independent processor units, with each processor unit containing its own local memory; there is no global memory. A fault in either processor or memory results in the designation of the entire unit as faulty. Each processor may read the memory of any of the processor units, but may write only to its own memory. Processor X can therefore not maliciously interfere with any other processor. Each processor's code is protected in its local memory and data may be checked as received. Like PLURIBUS, specially designed hardware is limited to

the bus system and interfaces between the busses and modules.

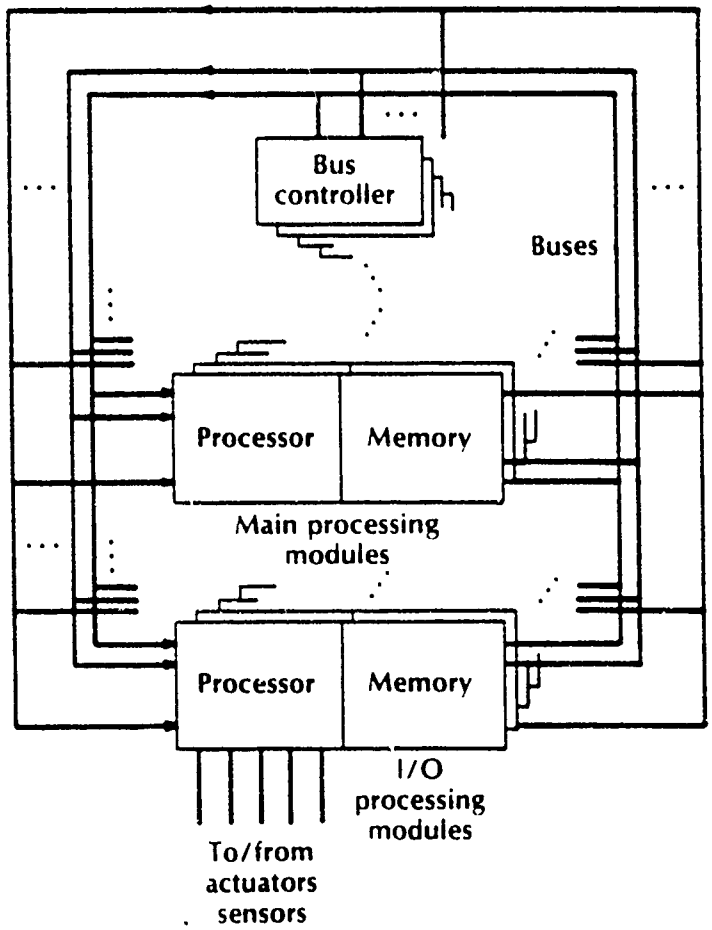

Figure 5. SIFT [93]

All software is designed as a sequence of *iterations*. Each iteration receives input from the previous iteration, and leaves any output in a memory buffer to be read by the following iteration. A particular software iteration will be simultaneously run on at least three processor units. So, when an iteration begins and reads its input data, it will read the data from buffers in at least three local memories where the implementations of the previous iteration were run. The iteration then votes on the input data; a difference results in leaving any error information in a buffer so

that a global executive can be informed of any suspected fault.

At this point, comparisons between FTMP and SIFT show both striking similarities and differences. FTMP also utilized TMR to mask single faults without need for program rollbacks. Spares can in both cases be assigned to replace failed modules. Prioritizing critical functions allows the system to remain operational despite degradation below full throughput capacity. A detected fault results in the removal of a module from the active system. Once removed, a module is considered failed for the remainder of the mission. Beyond these, characteristics diverge. FTMP handles faults with little time overhead. With SIFT, the software overhead to detect and recover from faults is continuous and must be included in calculating minimal required system throughput. Given that throughput is adequate, the SIFT approach demonstrates many advantages.

FTMP's hardware TMR approach meant that hardware modules must operate in lock-step, which requires fault tolerant clocking circuits. SIFT only requires that output from the triplicated iterations be available within some reasonable selected time window (such as 50 μs). Clock synchronization between processor modules is not nearly as critical. The processor units on SIFT do not have to be divided into threes with unused standbys. All SIFT processor modules may be active simultaneously, and each triplicated iteration may be executing on any combination of three. A standby processor module for a particular iteration must simply contain a copy of the iteration code in local memory so that it can quickly replace a failing processor module. The global executive, responsible for system configuration decisions, just like any other application software runs as triplicated iterations. The local executive at each processor module is

responsible for communication with the global executive. Communication can take place only by a processor reading the memory of another processor unit.

The advantages of software (easier version updates, improved error reporting, etc.) vs. hardware fault tolerance also applies to the SIFT vs. FTMP comparison. Significant software overhead and proof of correctness remain the foremost disadvantages.

It must be noted that in both PLURIBUS and SIFT the application software is special purpose. That is, the hardware configuration and system software are both designed to support a specific application, and, the application software is written to support a specific system architecture. It is easy to see that software fault tolerance as exemplified by these two systems ignores the difficulties involved in fault tolerance for systems designed to handle general purpose software. Thus one initial conclusion is that as the applications become less focussed, the fault tolerance becomes more difficult to implement in software and must necessarily migrate down into the hardware level.

2.3.2.3 UNIX RTR

UNIX RTR, unlike all of the hardware and software fault tolerant systems described so far, is a standard software system (UNIX) version. UNIX RTR is the operating system delivered with the AT&T 3B20 Duplex computer (3B20D). This system (see Figure 6) is designed for applications requiring ultra-high availability, both for communications switching systems and for general transaction processing systems [92, 96].

The 3B20D is a fully duplicated system. Each half of the system consists of a processor with local memory, attached disk controller, and

I/O processor. Each processor can access directly both its own disk controller and I/O processor and also the disk controller and I/O processor of its twin. One processor is active while the other is in standby mode. Each write by the active processor to its memory is also routed through the memory update controller of the standby processor, so that the contents of the two memories are identically maintained.

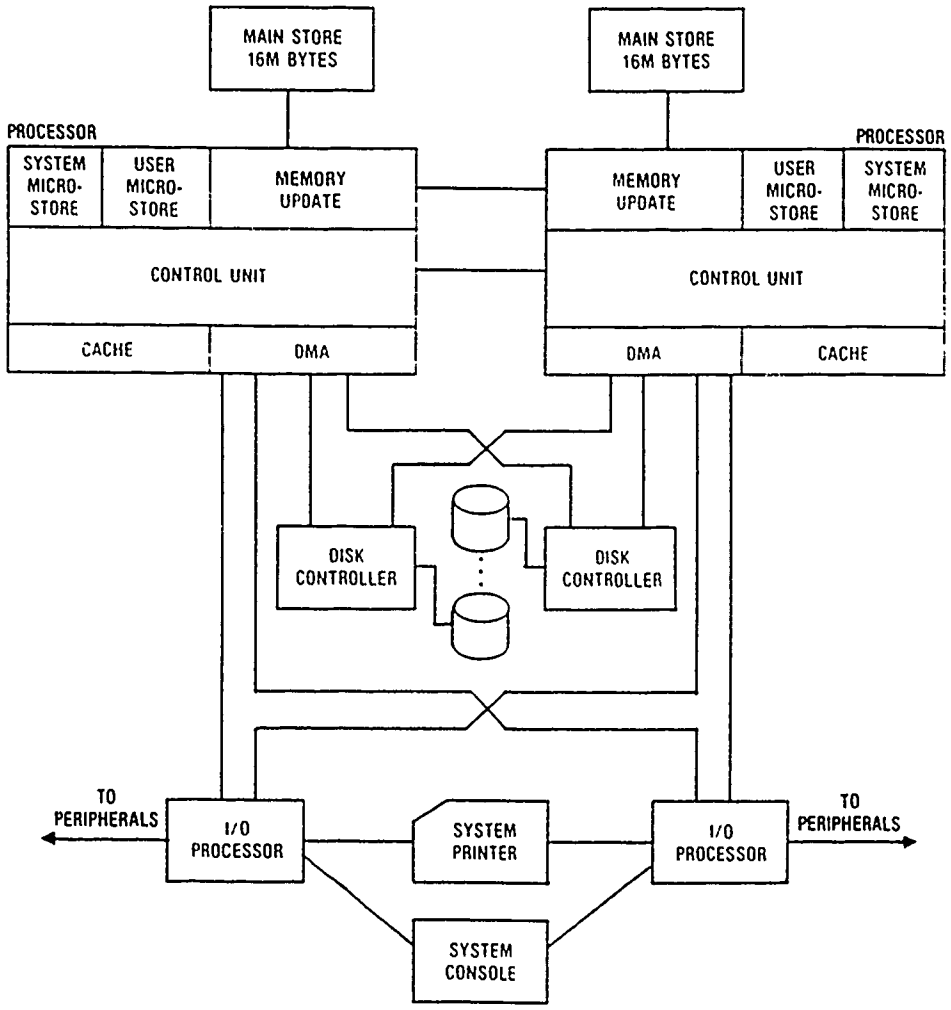

Figure 6. AT&T 3B20 Duplex Computer [92]

A detected fault in the active processor can be immediately masked by the standby processor by taking over as the new active processor. Disk

mirroring is a standard software feature; when activated for a given disk drive the output data is also sent to the backup disk drive, which is connected to the other processor's disk controller.

The philosophy behind the 3B20D is to provide ultra-high availability through the masking of any single fault (and some multiple faults) without any parallel duplicate processing. All decision making, including attempted recovery actions, is centralized in the active processor. Once a recovery level decision has been made, the device drivers are expected to be able to respond independently, in effect performing a distributed recovery. This allows peripherals to be added to the system as long as appropriate device drivers are provided.

Because the active processor is totally in control of the system (no TMR, shadow processor, or watchdog processor), the active processor must perform as much internal checking on itself as is practical. In addition, an on-line configuration database is maintained. This provides a description of hardware units, including both information essential to reconfiguration and also associated error counts and thresholds. Besides the normal hardware self-checks, UNIX RTR periodically performs a variety of software tests to check for reasonableness among system data structures. Allowing the standby processor to take over (with its mirrored memory) is appropriate only in response to certain faults. In the case of a corrupted system data structure, whichever processor is or becomes active must effect a recovery with minimal down time.

In order to minimize down time, UNIX RTR uses a *progressive recovery* scheme. Recovery levels are attempted in order of increasing expected down time. A local recovery (hardware switch or system data structure reconstruction) requires essentially no down time and will

always be first attempted. Next, a quick bootstrap may be necessary. If any current applications have called system routines that may not have completed correctly they must be so informed. Therefore, critical applications should include signal handling routines. A complete bootstrap, with the reloading of additional system software and the initialization of additional system data structures, would be the final recovery action that UNIX RTR would automatically perform. A complete reload, including memory clearing, requires manual operator intervention.

The UNIX RTR system integrity software periodically checks all major software systems. It monitors the operator interface, file system, maintenance control, software updates, and the system integrity software itself. In addition, interfaces are provided that allow application software to check itself, using the facilities of the system integrity software.

UNIX RTR provides extensive audits of hardware and software performance. Parameters describing relative criticality of applications are used for load balancing and handling system overload conditions. Allowance is made for updates to application software, operating system software, and hardware, all without requiring a system shutdown.

2.3.2.4 MACH

MACH [44, 53, 69] is another variety of UNIX with extensions. Unlike UNIX RTR, however, MACH runs on shared memory multiprocessing systems. It provides for multi-threading within a task's address space and can distribute threads amongst the available processors.

MACH is not UNIX. Rather, it has a completely rewritten kernel which re-establishes the original focus of UNIX on as basic a kernel as possible, containing only the most essential functions. However, MACH is

binary compatible with UNIX executable files. This is to allow for an easy upgrade path from standard UNIX environments to MACH. While support for multiproccessor systems and multi-threading within an address space are the most obvious distinctions of MACH over standard UNIX, in fact the rewritten kernel makes use of the current state of virtual memory systems [85] to increase efficiency and resilience also in single processor systems and networks.

MACH adds five abstractions to the UNIX environment [86]: task, thread, port, message, and memory object. These replace and/or extend standard UNIX concepts and are invisible to the UNIX programmer who chooses to ignore them.

A MACH *task* is roughly equivalent to an address space. A *thread* is a string of control; it can be thought of as a program counter and register set. In this context, a standard UNIX *process* would be a task with a single thread. Multiple threads require the existence of mutual exclusion primitives available both to the kernel and to application tasks. These same types of primitives are those required for multiprocessor scheduling. On a multiprocessor, separate threads within a single task may execute simultaneously on different processors.

The concept of a hardware communication *port* is generalized in UNIX to include logical ports which serve as interprocess communication end points. In MACH, ports are generalized further and assume an even greater role. Applications use ports to communicate with server tasks and also with objects managed by the kernel. Ports can move from object to object. Multiple threads may read from a port, but these threads must all be within the same task. Multiple tasks may write to a port. Special ports exist for certain specialized communications purposes. Notable among

these are the task exception port and the thread exception port. Exceptions are MACH extensions to the standard UNIX signal facility and will be covered separately.

The standard concept of a *message* has also been extended in MACH. Unlike UNIX, MACH messages contain user accessible headers in addition to the stream of bytes comprising the message body. *Simple* messages do not contain references to other ports; *non-simple* messages implement indirect port accessing. Messages in MACH are the primary way in which tasks communicate with each other and with the kernel. A message may be as large as the entire address space of the sending task. The kernel uses copy-on-write address translation to efficiently transmit the message by simply adjusting the receiver's address translation tables. Messages which are sent over networks between computers are handled transparently to the user, although in this case the entire message must be physically transmitted, so the efficiency of copy-on-write is not available.

The UNIX memory management concept is very basic – a *fork* system call results in the creation of a new address space. That's it; sharing memory between processes requires the processes to explicitly make shared memory request calls and manage memory contention problems on their own. MACH supports its inherent multi-threading and multiprocessing features through the creation and manipulation of *memory objects*. This concept allows each task to manage its memory on as low a level as it wants. For example, when forking a child task, the parent may specify which parts of the address space to copy to the child's address space. The remainder of the data in the parent address space is unavailable to the child. To implement copy-on-write for messages, the message becomes a

memory object in the receiver address space. Since threads share an address space, a task can create threads with much less overhead than forking separate tasks. And, it can still protect itself by restricting the created threads to certain memory objects while protecting the remainder of the task address space.

The significant improvement to memory management in MACH over UNIX is to provide a safer environment for application tasks. Another MACH feature central to the implementation of software fault tolerance is the exception handling facility [13]. In UNIX, application processes are notified of the existence of exceptions through *signals*. The kernel finds the exception and sets a bit in a 32 bit exception flag word associated with the process. As the scheduler selects the affected process to run next, it notices the flag and instead of restarting the process at its last execution point, it restarts the process it at the appropriate signal handler address. This signal handler executes in the task's address space. So, the signal handler has access to the data in the address space, but not to the saved context (which includes the registers and the program counter). And, since UNIX does not support multi-threading within an address space, the signal handler is severely restricted as to the actions it may take on the shared data structures. Basically, excessive effort must be taken to avoid access to critical regions by a signal handler routine since no convenient mechanisms exist that the task can use to lock the critical regions.

MACH exception handling takes advantage of the multi-threading feature. The exception handler is executed as a thread in the same task as the victim. Here the term *victim* refers to the task or thread to which the exception is sent. Victim context information is made available to the

exception handler thread. The handler thread has access to the victim's memory and the handler may either abort the victim or allow it to restart. If the handler has decided to allow the victim to restart, it may first send it messages through the victim's exception port. Signals are supported in MACH and may be used instead of or along with exception handlers.

Monitors and debuggers also take advantage of MACH extensions [49]. Under UNIX, a debugger must run as a parent process of the process being debugged. Otherwise, it could not be aware of signals to the debugged process. Once a process is running there is no way to attach a debugger to it; the debugger must have been running first so it could fork the debugged process. Under MACH the debugger can simply "take over" the exception ports from the debugged task and receive the exceptions and signals itself. These are subsequently passed on to the debugged task through a substitute exception port. So, the debugger has intercepted signals and exceptions to the debugged task. This allows the debugger to attach to and detach from the debugged process at will.

The IBM EPEX programming environment is designed to support multiprocessing application programs written in assembler, FORTRAN, and C. Originally written to run under IBM's VM/CMS control program, EPEX could not be ported to UNIX due to the lack of multi-threading within an address space and the difficulty of using shared memory. So, IBM has ported EPEX to run under MACH [15]. This programming system is called MACH/EPEX.

2.4 Distributed Systems

The next important type of system configurations is loosely coupled systems, which are logically and physically separated from each other

except for communication links. The idea is that each system is independently capable of stand-alone operations. The loose coupling adds the ability to distribute workload and/or data amongst the systems (which will be called *nodes*). The presence of duplicated systems, with intercommunication facilities, implies that this configuration can be adapted to fault tolerance.

2.4.1 Distributed Software

Reliability is but one of many issues considered during the design of distributed system software [83]. Other critical issues include task scheduling, decentralized control, protection and security, distributed file systems, etc.

One of the initial, yet important, uses for coordinated processing over a distributed system was for distributed databases. This increased efficiency as data could be maintained at the system where it was most likely to be accessed, while still available at other connected systems albeit with some access delay. Data did not have to be duplicated but could be accessed in sub-second intervals, adequate enough for query systems. The lack of duplicated data alleviated the inherent data consistency problem.

The data consistency problem does not disappear, just because there is no duplicated data. Anytime non-serial action is taken on data, whether in a loosely coupled system, a tightly coupled multiprocessor system, or even multi-tasking in a single system, the possibility of inconsistent data is present. Atomic actions and locking mechanisms common to multi-tasking systems must be generalized to the loosely coupled environment.

A *network operating system* (NOS) consists of communication routines to send messages between nodes [82], routines to handle file serving, distributed task scheduling, reconfiguration around node failures, etc. An operating system designed to manage the resources of a distributed computer system (DCS) in a global manner is called a *distributed operating system* (DOS).

The DCS software must account for the serialization of resources over a loosely coupled network [95]. Most approaches define certain resources whose access is limited to defined atomic actions. These resources can then be employed as locks in much the same way as resources are accessed by atomic read/write instructions within single processor and tightly coupled multiprocessor systems.

The loosely coupled network is further complicated by delays in receiving messages and increased likelihood of incorrect or missing messages, possibly due to node and link failures. Reliability requirements of distributed programs can be divided into two cases [76]:

(i) The program will not abort despite the loss of a finite number of messages.

(ii) The program will not abort despite the loss of a finite number of messages and of node failures.

Programs designed for operation in a distributed environment must emphasize resistance from aborts (or incorrect actions) to a far greater extent than programs designed for operation in a uniprocessor environment or tightly coupled system.

Assuring reliability using relatively unreliable communications in a distributed system encompasses a wide variety of issues. Recall the difficulties with processor agreement in the tightly coupled PLURIBUS

system. A generalization of these coordination protocols to a loosely coupled distributed computer system [59] illustrates the complexities. The development of algorithms which continue with correct operation during node failures, while simultaneouly minimizing performance degradations using unreliable communications, is an especially critical problem.

Fault tolerance in a distributed computer system at the operating system level generally emphasizes reconfiguration. One level requires that server processes be independent of the physical machine on which they are running. That is, instead of directly communicating with a server, a client may communicate with a directory service which dynamically forwards messages between the client and the server machine [32]. On a second level, the operating system is responsible for the decisions concerning the server (and client) reconfigurations. Generally this limits the applications to a special purpose system. The generalization of this key concept is essential to the FTM system, described in this dissertation.

2.4.2 Backup, Checkpoint, Recovery

Synchronization in a loosely coupled system is especially critical to ensure consistency for backup and checkpoint data. A detected error can prompt a checkpoint rollback in one or more of the nodes. In this case the state of the distributed system, after the rollback, must take into account the status of outstanding messages. If a message was sent from processor A to processor B and after rollback processor B has recovered to a point in time before the message was received, then processor A must also be rolled back to a point in time before it had sent the message, or at least to before it received an acknowledgement. The challenge is to assure consistency with less overhead than by taking a checkpoint with each message.

Kohler [45] surveys techniques designed to allow synchronized access to shared objects with a high degree of concurrency. The generalization of the concept of an atomic action to that of an atomic transaction is used as a basis for a *serializable transaction schedule*. Kohler contrasts five distinct concurrency control methods: locking, timestamps, circulating permits, conflict analysis, and reservation lists.

Locking an object is an extension to locking a resource in any multi-tasking system. A transaction locks an object after waiting for the object if it is already locked by another transaction. Before the transaction completes, it unlocks the object. Since transactions require many locks, tradeoffs between efficiency and deadlock prevention must be considered. This is especially critical on a distributed system because of the relatively slow communications times, and lost "unlock" messages (missing messages or node failures).

Timestamps (usually some function of a local clock) are assigned to transactions from a node in a monotonically increasing sequence. The node-timestamp identification on each message is used by the *concurrency controller* at the receiving node to synchronize access to objects. Protocols such as WAIT-DIE or WOUND-DIE govern the actions taken by the concurrency controller based on the relative timestamps of requesting transactions.

Circulating permits or *control tokens* are used to implement the token ring network. In this scheme a node must "own" the appropriate circulating permit before it may initiate a transaction. The circulating permit travels around the network, dropping off its message and receiving responses, until it is released by the original requestor. This scheme has many drawbacks. Besides the time wasted waiting for the appropriate

concurrency permit to arrive, the level of parallelism is decreased by the need to initially partition the data and preassign a concurrency permit to each partition. These partitions will in general have low granularity. Thus, there is the need to check for lost concurrency permits resulting from communication errors or node failures.

Conflict analysis is used to minimize synchronization overhead by pre-classifying the likelihood of interference between different types of transactions. The amount of locking is based on these likelihoods. This means that transaction rollback must be performed in those cases where insufficient locking has occurred. If this rollback occurs infrequently the average transaction delay is minimized and a higher degree of transaction parallelism is achieved. This scheme works only where transaction effects are well known, such as in a distributed database system. In more general systems the likelihood of interference cannot be sufficiently pre-classified to keep rollbacks to a minimum.

Finally, *reservation lists* are an attempt to combine several of the above described techniques. In this scheme each transaction must choose one of two available protocols. In one protocol, the transaction is guaranteed exclusive access to an object, but must wait until the object is available. The second protocol allows shared access for transactions. In this case the delay is likely to be shorter, but the transaction must be willing to restart, if necessary.

The maintenance of replicated data [34, 43], whether designed as a current remote mirror image or a periodically updated remote backup copy, is subject to similar synchronization problems as transactions with remote data. In the case of remote copies of data, the control of temporary inconsistency must be considered. Transactions may temporarily

leave data in an inconsistent state during processing, or longer if the transaction aborts. In this case rollback must restore data consistency. In the case of backup data the decision as to when to make a backup copy must be determined. If the rollback affects the remote backup copy, this copy must be placed in a state such that the data is consistent and coordinated with the rollback point of the transaction.

Each process, in a distributed system that uses rollback recovery, takes its checkpoint independently [46]. This asynchronous independent checkpointing will require a common consistent synchronized point, and coordination between all processes in checkpointing actions.

Another consideration is the possibility of a failure during the checkpoint. One approach to the coordination of checkpointing involves a two phase commit protocol. In this protocol, whenever a process decides to checkpoint, it informs all other affected processes (which in turn may inform others). All such processes attempt to take a checkpoint. If succesful, each such checkpoint is flagged as temporary. Once all processes have reported success to the originating process, the originating process broadcasts the universal success status and each temporary checkpoint becomes the permanent current checkpoint. This protocol coordinates the checkpoints and removes the need for further rollback. Thus, each process must maintain only one current checkpoint (or two between the creation of a temporary checkpoint and its designation as permanent).

2.4.3 Consensus

Consensus protocols can be used in a distributed system to identify faulty processors [9, 21]. A function is replicated at each of n nodes. Independently calculated results are then broadcast amongst the n nodes

such that each node can construct a vector recording its copies of the n results. A decision function is applied to the vector, and the system is considered "correct" if agreement is observed among the vector elements above some threshold number.

The safest consensus protocols insist that any processor detected as faulty immediately dissassociate itself from the distributed system. Failure models have been constructed under the assumption that processor faults occur independently according to some random distribution. While it may be that a failed processor was subject to a temporary aberration, unlikely to reoccur, the processor if allowed to continue may join with other failed processors in subsequent Byzantine behavior. The failure models have shown that immediate disassociation of processors with detected failures maximizes the time before accumulated processor failures remove the system from a reliable state.

2.4.4 Clock Synchronization

A typical loosely coupled network lacks a centralized clock-setting facility. Since each hardware clock will have its own rate of drift from real time, clock synchronization messages must be passed around the network to keep the individual clocks within some given time difference. Generally each system calculates a *logical time* by maintaining an adjustment to the time on its local physical clock. Clock synchronization algorithms are used to periodically set this adjustment.

Algorithms to synchronize clock adjustments fall into two categories, depending on whether or not message authentication is present and assumed valid [81]. The presence of message authentication precludes a serious type of Byzantine behavior where a system sends

incorrect synchronization messages while assuming the identity of a different system.

Two assumptions are basic to clock synchronization algorithms:

- 1. The rate of drift for each physical hardware clock is linear. Let $R_i(t_j)$ be the reading of hardware clock i at time j. Then: for some $\rho>0$:

  $$(1+\rho)^{-1}(t_2-t_1) \le R_i(t_2)-R_i(t_1) \le (1+\rho)(t_2-t_1)$$

  Logical clock i is said to be within a *linear envelope* of the real time t.

- 2. There is an upper bound on the time for a message to be processed by the receiver after being prepared and transmitted by the sender.

It is easy to see that no synchronization algorithm is possible without assumption 2, since the time delay from message transmittal until message receipt and processing would be arbitrary. Assumption 1, requiring linear physical clock drift, is not a necessary condition for the existence of synchronization algorithms. But, it greatly simplifies the analysis of optimal algorithms and is not an unreasonable restriction.

A synchronization algorithm must satisfy two conditions. The *agreement condition* is that all correct logical clocks agree on the time within a given allowable deviation. The *accuracy condition* states that all correct logical clocks are within a linear envelope of the real time. Note that it is not possible to do better than a linear envelope for the accuracy condition, since no tighter assumption for each physical clock was made. Fortunately, the critical time element for synchronization is the agreement

condition.

With these two assumptions, clock synchronization algorithms have been described which perform correctly as long as more than $^2/_3$ of the systems are non-faulty. If accurate message authentication can be assumed, then it is possible to employ algorithms which require only that more than $^1/_2$ of the systems be non-faulty.

As with virtually any model, a loosening of the assumptions allows for algorithms which handle more general cases. Efficient synchronization algorithms need make no assumptions about the minimal required proportion of non-faulty processors, as long as the sub-network containing the non-faulty processors remains connected [22], if another Byzantine scenario can be ignored, in which faulty processors act in concert by synchronizing themselves to a common incorrect time.

Most clock synchronization algorithms allow for initial start-up network synchronization, but do not allow for the addition of unsynchronized processors to an already synchronized network. This task is left to a separate algorithm designed specifically for this function.

2.4.5 Distributed Operating Systems

The control programs of distributed systems can by separated into three classes [83]: network operating systems (NOS), distributed operating systems (DOS), and distributed processing operating systems (DPOS).

If each of the hosts contains a fully functional local operating system, then the operating system software added to these local operating systems in order to communicate and share network resources is called a *network operating system*. A NOS is built on top of an existing operating

system. It adds the capability to support distributed transaction processing [80]. Besides the usual communications functions (message passing, file transfer, network virtual terminal, etc.), distributed transaction processing requires facilities to support data replication, atomic actions across nodes, system wide time synchronization, and security for remote data access.

In contrast to a NOS layered onto a fully functional local operating system, a *distributed operating system* controls a network where there is logically only one native operating system for all the distributed components. The DOS manages all the resources of the network in a global manner. There is a wide variety of DOSs.

A *distributed processing operating system* takes the concept behind the distributed operating system one step further. Control is decentralized, thus improving system reliability by eliminating a central point of failure.

Each of these operating system classes provides for communication between the distributed sections of jobs. The actual control of the distribution may be defined by each job itself, or the distributed operating system may attempt to load balance. Load sharing policies in a distributed environment [74] require much more information than would be needed in a tightly coupled multiprocessor environment.

Even in a shared memory multiprocessor system, there are advantages to the use of a distributed operating system [16]. This allows the multiprocessor to consist of asynchronous processors designed and configured specifically to optimize the system under the expected workload.

2.4.5.1 LOCUS

LOCUS [63], a distributed operating system, like MACH, is not UNIX but emulates the UNIX environment. While MACH's generalized concept of ports is used to somewhat enhance network transparency, the main thrust of MACH is the addition of multi-threading and multiprocessing to a UNIX setting. LOCUS, by contrast, has network transparency as its principal goal. In a high bandwidth local network environment, LOCUS presents network transparency while remaining application code compatible with UNIX and uses the required network management routines to improve efficiency and fault tolerance. LOCUS is not a distributed processing operating system because each node can run independently and retains as much local autonomy as possible. LOCUS does manage the replication of data amongst the nodes and provides extensive synchronization routines to efficiently handle data accesses in this distributed environment.

Walker and Popek [91] discuss IBM's implementation of LOCUS, the Transparent Computing Facility (TCF) for the AIX operating system. AIX is IBM's version of UNIX which runs on IBM RT workstations. In this discussion, they define *network transparency* as the combination of six transparency components:

1. *Access transparency* means that the same system calls are used to access files, devices, processes, and interprocess communications entities, regardless of the location of the particular resource.
2. *Device transparency* is a subset of access transparency. It is listed separately because access transparency is sometimes interpreted to refer solely to data file access.
3. *Process transparency* is also a subset of access transparency.

This is because process transparency is always present in access transparency (as part of remote mount distributed file systems). Listing process transparency separately is meant to imply a more general concept where a full complement of processes are remotely accessible.

4. *Location transparency* means that resources do not have a site's location built into their name. This allows resouces to be replicated or relocated.
5. *Consistent file access semantics* are necessary. Preserving file access, modification, and synchronization semantics over processes executing on several different machines is required for the processes to be able to cooperate.
6. *Performance transparency* means that the overhead involved when remote resources are accessed exceeds corresponding overhead for local resources by a small enough amount that the additional overhead can be ignored.

The main goal of LOCUS is to make the development of distributed applications no more difficult than the equivalent single machine applications, and to employ the distributed environment to maximize the potential for highly reliable and available operation. The major assumption is that the machines in a LOCUS network are interconnected with a high bandwidth, low delay, low error rate communications medium (such as ethernet). All sites in this network are to run LOCUS, so that there is no need to support standard protocols (such as TCP/IP or Decnet) between sites in the network. The processors in the LOCUS network may vary widely in power and storage capacity.

Network transparency means that applications need not specify a specific site for any needed files or services. Locating and accessing the data or service is entirely within the operating system code. Processes have associated *reliability profiles* which are used by LOCUS to determine optimal replication of resources for the application. This replication may include data for the application, required services, and copies of the application code itself. LOCUS attempts to maintain operation in the face of errors, including even node failures and partitioning of the network.

Since data and services may be transparently distributed around the network to users, the names must be network wide unique. The applications need not know the current location of the desired data or service, so the unique name may not contain a field specifying any particular node. And, since data and services may be duplicated, an application should be dynamically connected to the most efficiently accessed instance of possibly several currently available at several nodes. For example, if file "X" has been replicated by the system and exists at nodes A and B, then an application requesting access to file "X" should be connected to the system with the best expected response time. But, since names must be network wide unique, A and B cannot both have a file named "X". The LOCUS solution is to tag each name with a system identifier. The application may invoke a low level selection by specifying the tag, or it may allow the tag to default, in which case the system on which the application is running will choose the tag from its current knowledge of network wide names.

Tags actually contain considerably more information than current system identifier. For example, tags on binary files can indicate which systems support the necessary run-time environment. This is so that the

Network transparency means that applications need not specify a specific site for any needed files or services. Locating and accessing the data or service is entirely within the operating system code. Processes have associated *reliability profiles* which are used by LOCUS to determine optimal replication of resources for the application. This replication may include data for the application, required services, and copies of the application code itself. LOCUS attempts to maintain operation in the face of errors, including even node failures and partitioning of the network.

Since data and services may be transparently distributed around the network to users, the names must be network wide unique. The applications need not know the current location of the desired data or service, so the unique name may not contain a field specifying any particular node. And, since data and services may be duplicated, an application should be dynamically connected to the most efficiently accessed instance of possibly several currently available at several nodes. For example, if file "X" has been replicated by the system and exists at nodes A and B, then an application requesting access to file "X" should be connected to the system with the best expected response time. But, since names must be network wide unique, A and B cannot both have a file named "X". The LOCUS solution is to tag each name with a system identifier. The application may invoke a low level selection by specifying the tag, or it may allow the tag to default, in which case the system on which the application is running will choose the tag from its current knowledge of network wide names.

Tags actually contain considerably more information than current system identifier. For example, tags on binary files can indicate which systems support the necessary run-time environment. This is so that the

network may contain processors with different architectures and different nodes may contain different local peripherals and drivers. Again, the aim of LOCUS is to provide default tags which are based on a system understanding of the required resources, rather than requiring the user to explicitly defined appropriate tags. LOCUS, like UNIX, uses the file system hierarchy to define names for services along with data file names.

Because file names, including path information, are visible to UNIX applications, the high level view of the LOCUS file system emulates the UNIX directory tree structure with all names unique across the network. The low level view of the LOCUS file system also maintains names that are globally unique. In UNIX, the file system is partitioned into *file groups*. Each file group consists of a set of file descriptors and a (much larger) set of data blocks. So, a low level file name is described as: <file group number, file descriptor number>.

In LOCUS, since file groups can be replicated, the UNIX file group number becomes a LOCUS *logical* file group number. There is an additional mapping from the logical file group number to the physical file group number. Since logical to physical can be a 1-to-many mapping, these map routines choose the optimal physical file group. Note that this means that a file can only be replicated at a node which contains the file's physical group. However, the presence of a physical file group at a node does *not* imply that all files in that file group must be present at this node.

The replication of objects, both data and services, across the network mandates that LOCUS provide synchronization mechanisms far beyond those employed by standard UNIX. Recall that MACH employed a variety of synchronization services within a single shared memory system. LOCUS synchronization mechanisms emphasize distributed object locks.

Because LOCUS was designed for efficiency in a high-bandwidth local network, synchronization policies must also aim for efficiency – holding distributed locks for as short a time as possible.

The general synchronization approach taken by LOCUS is the "multiple readers, one writer" policy. Write access to an object (such as a file) locks it from other write accesses. Replicated copies must be updated or invalidated. If a replicated copy is invalidated, then current read accesses must be redirected transparently to the reader. The copies can then be re-replicated once the writer has closed its link.

This policy is not always sufficiently granular. For example, when updating a file it is also necessary to have write access to the containing directory. It is not reasonable to lock the containing directory for the entire time that one of its contained files is open for writing. On the other hand, implementing locks at the record level would entail high overhead and would be incompatible with standard UNIX (as it would be visible to applications). An application could gain access to a file but be unexpectedly blocked from a certain record. To get around this policy as gracefully as possible, LOCUS defines a *nolock read.*

The nolock read ignores any write lock on the file. Instead, it locks the file itself, but only for the duration of the current read. This means that if the file can be written in multiple steps, then a nolock read between the steps would result in the access of inconsistant data. Directories in LOCUS are updated by atomic writes, so the nolock read is used as the solution to the directory access problem. Other objects limited to atomic writes may also be read correctly and efficiently by nolock reads.

For each logical file group, one site is designated as the *current synchronization site* (CSS). The CSS coordinates all access (except nolock reads) to the files in its group, ensuring that the requestor receives the latest version of the file. It is not necessary for the CSS to contain locally any of the physical files in that group; any site may serve as CSS for any file group.

The designers of LOCUS approached reliability considerations as part of four broad classes:

1. The ability to substitute alternate versions of resources to replace a flawed original. This is addressed by the replication of objects, including files and services, and their transparent access across the high-bandwidth local network. In the case of a resource failure, LOCUS attemps to handle the substitution at as low a level as possible, signalling the application only in the case where transparent substitution is not possible.

2. *File committing* is used to ensure atomicity of multiple file updates where this is necessary. Commit is automatic at file close time, but applications may also request commit at any time between single updates. Commit causes the changes to be made permanent at the site of the physical file, followed by propagation of the new version of the file in parallel to other storage sites.

3. LOCUS is a distributed operating system (DOS), but not a distributed processing operating system (DPOS). This means

that even if a site is completely isolated from the network, it can continue useful work with the services and data locally available.

4. The interaction between the machines is designed to promote "arms length" cooperation. That is, as the LOCUS operating system receives messages from other machines it performs consistency checking against its view of the current state of the network. LOCUS distributes resources, but not control, so it attempts to prevent illogical messages from corrupting its view of the network.

Network topology changes (sites entering or leaving the network) are handled by LOCUS at a low enough level to be transparent (if possible) to standard applications. As with other lost resources, the application is signalled only if transparent continuation of operation cannot be accomplished. The topology change procedure runs a simple algorithm at each remaining site; this algorithm forges consensus on a coordinating site which then accomplishes the actual topology change. The coordinating site polls other sites to reconstruct network wide data structures and broadcasts copies of the reconstructed structures. If a deleted site was the CSS for a file group, the coordinating site chooses a different site to becoome the CSS. The new CSS must then synchronize information relating to that new file group. It invokes the file group recovery to ensure replicated file consistency and decides on further replication sites.

It was previously stated that LOCUS supported continued operation even if the network becomes partitioned. A partitioned network allows multiple writers: one per partition on instances of the same file.

The problem arises when the partitions are joined. LOCUS attempts to support this by defining a *version vector* which is maintained with each copy of the replicated data object. After join, as the data objects are merged, the version vectors are compared to detect conflicts. On certain data objects which are managed by LOCUS and are restricted to *add* and *remove* operations (such as user mailboxes and file directories), LOCUS can resolve conflicts. In other cases, conflicts must be reported to higher layers (data management routines or, as a last resort, to the application) which may be able to resolve the conflict.

LOCUS does not follow the UNIX (and MACH) objective of minimizing the kernel by restricting it to such basic functions as dispatching and address space manipulation. Rather, LOCUS maximizes efficiency by including common system routines in the kernel. This saves the overhead of process creation to handle these common services.

Kernel routines are implemented through kernel support for lightweight processes called *server processes*. A server process does not have its own address space; all its code and its stack are present in the kernel permanent storage. Server processes call other operating system routines directly. Server processes are linked directly into the kernel and remain static until the system is reinitialized with an updated version of the kernel. Network requests are handled by server processes. This, along with the required high bandwidth network, allows LOCUS to handle the volume of communications required to support replication of objects across the network much more efficiently than if it followed a standard network layering concept such as OSI.

LOCUS uses the current configuration data to optimize operations. Access to a file which exists only at the local site and whose CSS is local

(true except in unusual circumstances) results in the bypass of virtually all network routines. The cost is a slight additional complexity in protocols.

2.4.5.2 MEDUSA

MEDUSA [60, 79] is a distributed processing operating system (DPOS) designed specifically to run on the Cm* hardware. MEDUSA was not designed simultaneously with Cm* but rather as a replacement for StarOS, which was the original Cm* operating system. The Cm* hardware, with its connection of many low capability processors, is a testbed for distributed operating system design. The interconnecting modules (Kmaps) are microprogrammable, and in fact should be microprogrammed specifically to support the communications protocols and memory mapping needed by the current operating system.

As can be seen in Figure 7, Cm* consists of a number of computer modules (in this case 50) grouped into a number of clusters (in this case 5). Each cluster consists of a number of computer modules and a Kmap, with a single bus interconnecting all the computer modules within a cluster and the cluster's Kmap module. The Kmap modules are then connected to each other by a dual (redundant) bus. There is nothing magic about the number of computer modules per Kmap or the number of Kmaps, as long as the microprogrammable Kmaps can be programmed to handle the desired configuration.

Each computer module (cm) is a Digital Equipment microcomputer (LSI-11) with 64K or 128K bytes of memory. The LSI-11 has hardware support for two address spaces. Typically address space 0 contains an operating system (in this case the MEDUSA kernel) and address space 1

for user tasks. The Kmaps are special purpose communication controllers, microprogrammable, with 4K of 80-bit microprogram store per Kmap. There are no hardware precedence levels between Kmaps or between cm's. This means that there is no central authority in the system; all coordination must be software enforced.

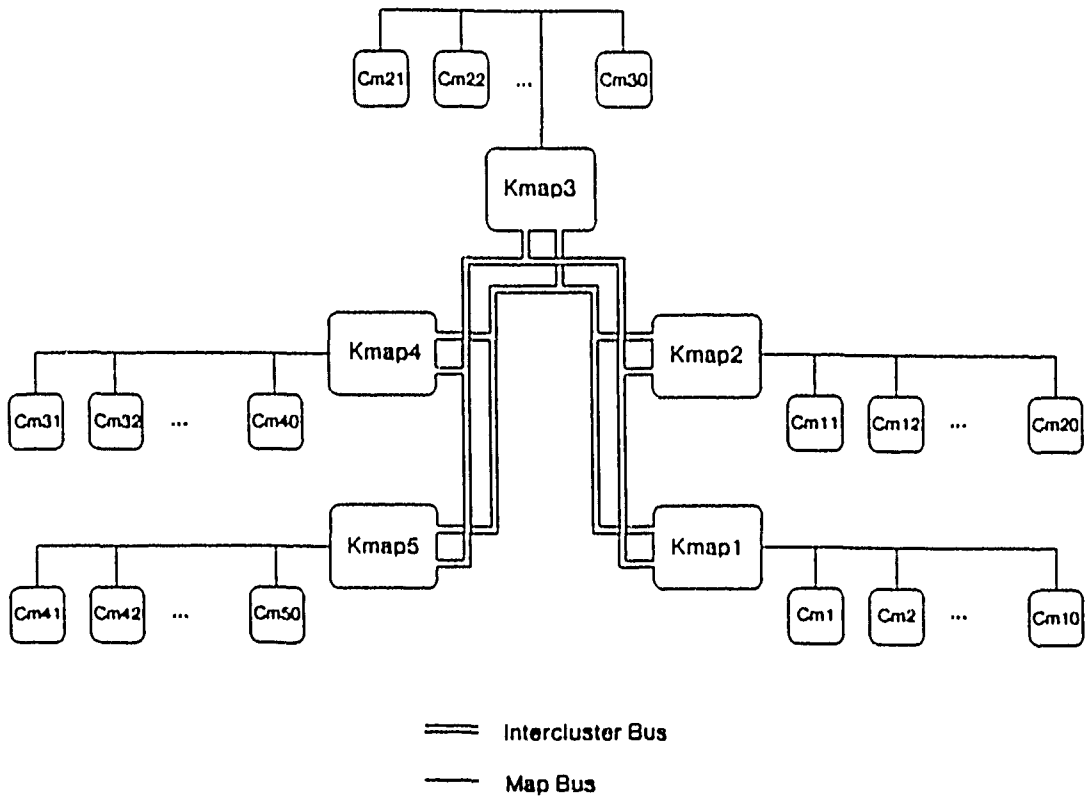

Figure 7. Cm* Hardware Structure [60]

From Figure 7, Cm* appears to be simply a network with dual connections between bridges. An examination of Figure 8 shows that each cm (an LSI-11) has actually been modified by the addition of a module called a Slocal. This Slocal has been inserted between the LSI-11 processor and the memory bus. As is true with the PDP-11, the memory bus is actually a general purpose bus which accesses both memory and peripheral I/O

devices through the same local address space. Any memory address referenced by the LSI-11 is sent to the Slocal, which decides whether to route the request to the local memory space or instead to send an appropriate message to the cluster Kmap over the map bus.

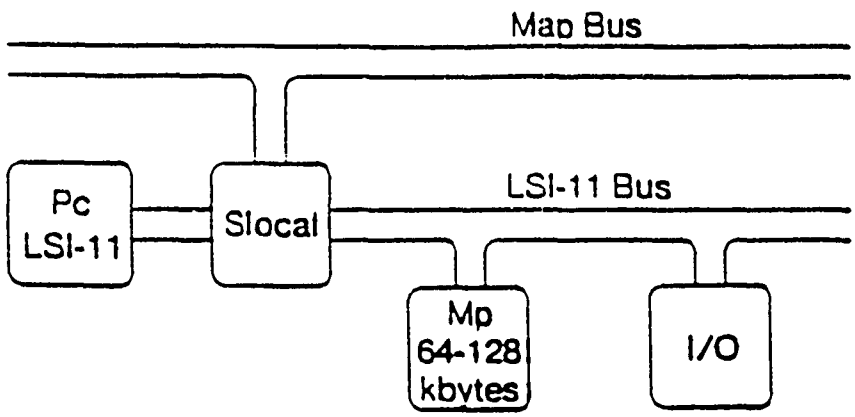

Figure 8. Computer Module (cm) Organization [60]

The general concept of Cm* should now be clear. As each LSI-11 accesses a memory location, the Slocal checks to see if the address is available in the local address space. If so, the data is fetched with minimal delay. If not, the Slocal must format a message and send it to the cluster Kmap. If the requested data is available within the cluster, the Kmap will send a message to the appropriate Slocal, which will send a message including the data to the requesting Slocal. If the requested data is not available within the cluster, then the Kmap must send a request by message to the appropriate other Kmap, which sends a message to the appropriate Slocal within its cluster, etc.

Despite the fact that the sole purpose of the Kmap is to facilitate Slocal to Slocal communication, there is a significant delay involved in the

access of non local data. Average access times in a lightly loaded Cm* system running MEDUSA are: 3.5 μs for local reference, 11.2 μs for non-local reference within a cluster, and 30.5 μs for inter-cluster reference. Maximizing system efficiency can be seen to be virtually synonymous with maximizing the local hit ratio.

The Slocal is a hardware device which operates off the memory address bus. This means that an LSI-11 processor could actually be executing a program whose code is not present in the local memory. It is hard to imagine when this might be desirable. Since an LSI-11 has no cache memory, execution efficiency would be low. The Cm* configuration makes the possibility of upgrading the processors to models with cache memory minimal since cache coherence would grossly complicate the responsibilities of the Slocals and Kmaps and destroy the emphasis on minimal delay memory references and message passing.

MEDUSA consistes of three distinct parts: the MEDUSA kernel (which runs at each cm), the utilities (which are distributed), and the supporting Kmap microcode. The emphasis within the Cm* project is the study of approaches to distributed processing as an alternative to a much larger single machine. That is, how can problems be solved more quickly (and/or at lower cost) in a distributed system. Fault tolerance is a secondary consideration. In fact, the only hardware devoted to fault tolerance is the dual interconnection bus between the Kmaps. However, the nature of the massively replicated hardware, although designed for parallel processing, allows the software a foundation from which to include fault tolerance principles.

The fundamental unit of control in MEDUSA is the *task force*. A task force is a collection of concurrent activites. Each activity is a task

which runs on a single processor. All programs in MEDUSA, including the system utilities, are task forces. A task force may have only one activity, in which case it is the equivalent of a single task on a uniprocessor. However, the low individual capacity of each LSI-11 cm means that many programs, including most system utilities, must be run as multiple activity task forces. The individual activities communicate and coordinate through messages. Recall that the sole purpose of the microprogrammed Kmaps is to provide efficient message passing services.

In addition to its activities, a task force contains a collection of objects to be accessed and manipulated by these activities. These objects are divided into three classes: page objects, pipes and semaphores, and control data structures.

A page object is simply a 4K portion of some activity's address space. Recall that while a task force may encompass many cms, each activity is a single task on a single cm. The LSI-11 supports a 64K address space for a task; this is composed of 15 page objects (4K each), with 4K reserved for data associated with communications between the cm and the cluster's Kmap. An access of non-local data may result in the transfer of a small data item or the transfer of the entire enclosing page object. Kmaps and Slocals are optimized for transfers of up to a full page object in a single message.

Pipes and semaphores are the message passing and synchronizing structures used in MEDUSA. They are implemented primarily in Kmap microcode, with kernel routines to map into the protocols. The internal representations are not directly accessible to the LSI-11 programs (including the kernels). This both protects the Kmap (which aids in isolating aberrant behavior) and also allows transparent Kmap microcode updates.

Control data structures include file control blocks, task force control blocks, descriptor lists, etc. These are implemented mainly by the operating system utilities. Application programs send messages (through pipes) to these utilities in order to perform operations on control data structures.

Each cm runs a MEDUSA kernel in its address space 0. The kernel's sole purpose is to supply simple multiplexing between the device drivers (present in address space 0) and the user and utility activities allocated to this processor (present in address space 1). Even decisions regarding priorities among activities are left to the system utilities. The kernel responds only to hardware interrupts and messages from the system utilities. User activities are not granted direct communication privilege; they must communicate with the kernel through the system utilities.

All kernel memory is local. The kernel never accesses a non-local address; this would result in a Slocal message to the cluster Kmap which means that in case of catastrophic Kmap failure the kernel would be non-operational. So, each cm is isolated from external failures. An isolated cm is ordinarily not very useful in a highly distributed system such as Cm*, but at least the kernel will remain operational so that it can shut down gracefully after saving some termination information.

Since the MEDUSA kernel is minimal, most of the operating system functions are provided by the system utilities. These system utilities are given privileges not available to user task forces, such as the ability to communicate with the kernel. There are five utility task forces, each implementing several abstractions for the rest of the system. These five are: the memory manager, the file system, the task force manager, the exception reporter, and the debugger/tracer. Activities within the utility

task forces share no common memory; shared data is distributed through messages. In this way, utility activities are in no way restricted as to distribution amongst computer modules.

MEDUSA pipes are similar to those of UNIX, where a pipe is a connected socket pair. Because MEDUSA uses pipes exclusively for communication with the operating system (UNIX uses subroutine calls and kernel traps for most operating system user interfaces) a MEDUSA pipe cannot be quite as unrestricted as in UNIX. Messages in UNIX are simply streams on uninterpreted bytes. MEDUSA maintains the concept of an uninterpreted stream, but adds control information as to message source and message boundary. In UNIX, the fact that it is possible to read, for example, 10 bytes from a pipe could mean that there is a 10-byte message, or 10 1-byte messages from as many as 10 different senders, or anything in between. Since system utilities in MEDUSA rely exclusively on multi-sender pipes to communicate with user activities, a malicious user activity could make it impossible for the utility activity to accurately separate out the malicious messages. Sender information and message boundaries allow for the needed discrimination between messages in the received byte stream.

While the data in the pipes can be interpreted by the receiver in any way, the MEDUSA system is designed to optimize "pass by value" messages. A pointer may be passed, and when the pointer is used by the receiver to access data, that data will be made available by the Slocal. Since pointers are frequently to large structures, only a small subset of which will actually be accessed, this minimizes actual data transfer. However, since the Kmaps are optimized for transfers of up to a page object (4K) it is usually preferable to transfer the actual data rather than a

pointer and thereby avoid the subsequent transfers of the needed individual data items by separate messages from the Slocal.

As was mentioned, the only hardware specifically included for fault tolerance is the dual inter-Kmap connection bus, although by its distributed nature the hardware replication provides obvious support for fault tolerance through software replication. MEDUSA also emphasizes efficient distribution of processing, although through its control over distributed software with location transparent communications it provides a solid foundation for fault tolerant applications. MEDUSA implements fault tolerance on the operating system level through the replication of each utility task force

The general approach supported by MEDUSA, to allow software monitoring and migration, is called the *buddy mechanism*. An activity can designate any other activity in its task group to be its buddy. Any exception not processed by the original activity is then passed on to its buddy. This can be supplemented by periodic communication between the original activity and its buddy. The buddy may be a standby activity whose sole purpose is to back up the original activity, or it may have a function of its own. Ordinarily the buddy would run on a separate cm; in any case the buddy is allowed access to the entire address space of the original activity. (In some cases, such as cm hardware failure, no original data may be available). The system utilities use the buddy mechanism in case of failure to detect the failure and move communication pipes from the failed activity to its replicated copy.

2.4.6 Distributed Programming Languages

Distributed programming languages provide high level mechanisms

for process communication and synchronization. The categories of distributed programming languages are [83]: concurrent programming languages, message passing languages, remote procedure languages, and hybrid languages.

Concurrent programming languages include Mesa, Modula, and concurrent Pascal. These languages provide for the interaction between processes via the sharing of resources, including variables in shared memory. This is an appropriate approach only for a multi-tasking uniprocessor or shared memory multiprocessor environment.

Message passing languages include CSP, Gypsy, PLITS, Guardian-extended CLU, and Smalltalk. These languages use mailboxes and messages for interprocess communication, rather than shared resources. This approach is less efficient in a shared memory environment, but has the advantage that it can also be implemented in a loosely coupled network environment.

Remote procedure languages include Distributed Process, Argus, and Ada. Messages are still used for interprocess communication, but the interface to the user is strictly that of standard procedure calls; the details of the actual message passing are hidden from the user by the procedure call interface. The user therefore need not be concerned with the underlying message passing protocol.

Hybrid languages include *MOD and SR. As the name implies, these languages implement both shared resource and message passing communication facilities. The idea is to be able to tightly bind processes sharing memory while also allowing communication (less efficiently) between remote cooperating processes.

for process communication and synchronization. The categories of distributed programming languages are [83]: concurrent programming languages, message passing languages, remote procedure languages, and hybrid languages.

Concurrent programming languages include Mesa, Modula, and concurrent Pascal. These languages provide for the interaction between processes via the sharing of resources, including variables in shared memory. This is an appropriate approach only for a multi-tasking uniprocessor or shared memory multiprocessor environment.

Message passing languages include CSP, Gypsy, PLITS, Guardian-extended CLU, and Smalltalk. These languages use mailboxes and messages for interprocess communication, rather than shared resources. This approach is less efficient in a shared memory environment, but has the advantage that it can also be implemented in a loosely coupled network environment.

Remote procedure languages include Distributed Process, Argus, and Ada. Messages are still used for interprocess communication, but the interface to the user is strictly that of standard procedure calls; the details of the actual message passing are hidden from the user by the procedure call interface. The user therefore need not be concerned with the underlying message passing protocol.

Hybrid languages include *MOD and SR. As the name implies, these languages implement both shared resource and message passing communication facilities. The idea is to be able to tightly bind processes sharing memory while also allowing communication (less efficiently) between remote cooperating processes.

2.4.6.1 PLITS

PLITS (Programming Language in the Sky) [26] adds three constructs to the normal programming language environment: modules, messages, and assertions. A module shares no information with other modules (there are no global variables) and modules communicate with each other only through the exchange of asynchronous messages. Assertions are employed to allow the PLITS compiler to optimize consistency checks.

First, some PLITS definitions. Note that PLITS tends to redefine common data processing terms rather than create new ones, which can be confusing. A *distributed job* (DJOB) is a collection of modules cooperating with each other in a single application. The DJOB may be distributed over several network connected systems (see Figure 9). Modules within a DJOB at a single system which communicate mainly with one another or share code may be grouped together; they then become a *site*. Any DJOB may have multiple sites at a single system and/or at multiple systems. Each DJOB has a single module called the *controlling module*. The controlling module initiates and terminates the DJOB. It is also responsible for taking appropriate action if one of the other modules in the DJOB fails.

Each site includes a collection of routines called the *kernel*. Recall that each system will in general have multiple sites, so these site kernels are in addition to the underlying operating system kernel. The site kernel implements multi-threading and message passing amongst the modules within the site. There is a single *Host Control Program* (HCP) per system. When a kernel at a site needs to exchange messages with other sites it communicates with the HCP. The HCP forwards the messages, either to

the appropriate site kernel at the local system or the HCP at the appropriate remote system.

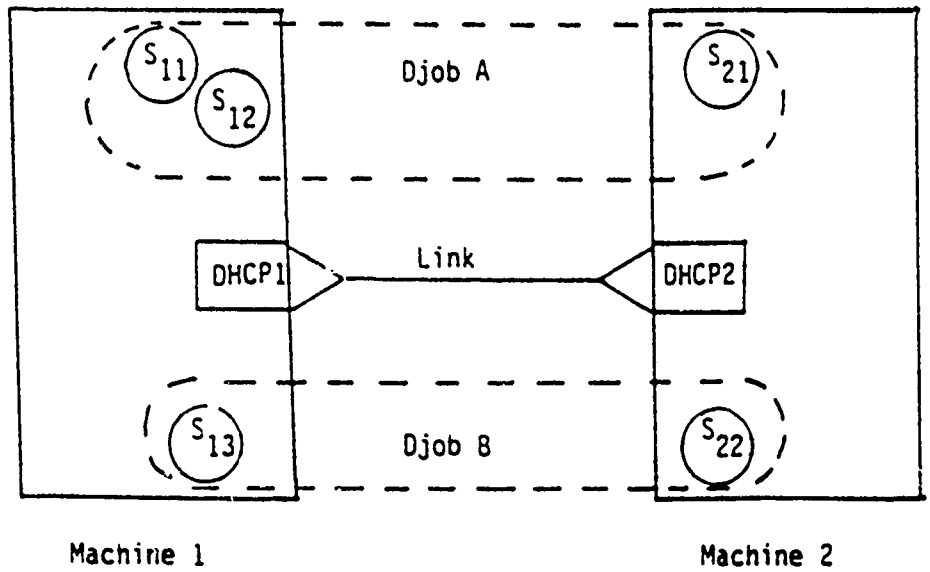

Figure 9. PLITS Distributed Jobs [26]

A message in the PLITS environment is highly structured. It consists of a sequence of (name, value) pairs, called *slots*. PLITS enforces typing on the value portion of each slot. The HCPs automatically perform conversions as required. For example, different system might differ in their representations of floating point numbers. This implies that all modules within a site must support the same primitive data types, since the HCP is not involved with communications within a single site.

The PLITS commands to send and receive messages are, naturally enough, SEND TO and RECEIVE. RECEIVE may be blocking or non-blocking. The parameter may be a predefined message (think of a message as a structure of slots) or it may be a sequence of variable names (from which PLITS will construct a temporary message.) In either case, the

name portion of each slot must have been declared as *public* so that typing information will be present in the header to the executable program file. The names within a message must have been declared public in both the sending and receiving module so that they can be compared as to type by the site kernels (for communication within a site) and by the HCP (for intersite communication with possible conversion).

Selective receives are optional. A client who has sent a request to a server may wish to receive a reply before processing any other incoming messages. So, a specific sender may be specified with a RECEIVE request. A more general option is also available: a SEND request may include a request for the site kernel to assign a unique *transaction key*. The module may then selectively RECEIVE a return message with that particular key. This allows the server to pass on the request to a third module, which can answer directly rather than having to return the message through the original server. The server, and any third module it may use, must record the transaction key and send the same key as part of its return message.

An *assertion* is a predicate which is guaranteed to be true at run time. Assertions are used to check for consistency during program executions. These have two basic forms: ASSERT (assertion) which generates an exception if (assertion) is not true, and UNDER (assertion) DO (statement) which conditionally executes (statement). The assertion mechanisn does not add any power unavailable through standard "if" constructs. Its purpose is to allow the compiler to employ artificial intelligence (AI) techniques for run time optimization. The idea is to take each assertion – a sequence of disjunctions and conjunctions of predicates – and reduce the assertion to its simplest form. In some cases, the result will be a constant

so that all the work will have been done at compile time. In other cases, the simplest form will hopefully entail much less run time overhead than the original, which will encourage programmers to employ assertions more often than they might otherwise.

PLITS is mostly an *approach* to distributed computing. Implementations have always been subsets of the full PLITS standards, in academic settings. Fault tolerance is minimal: messages between modules are typed and optionally tracked (with transaction keys), programmers are encouraged to use assertions liberally to check for illogical states during program execution, and distribution of sites across systems allows alternate hardware configurations. But, runtime module relocation is not supported, a single control module is responsible for each distributed job, and virtually any module failure (site kernel module, HCP module, user module) can bring down the job or the entire PLITS environment.

2.4.6.2 ARGUS

The basis for the ARGUS integrated programming language and system [50] is to address that class of applications in which the manipulation and preservation of long-lived data is central. These applications include banking systems, airline reservation systems, office automation systems, and data base systems. ARGUS provides constructs and facilities in the context of a high level language. These constructs and facilities allow for the distribution of the data and the processing requirements amongst the nodes of a loosely connected network.

The intended application class, along with the assumed loosely connected hardware environment, leads to several requirements: service, extensibility, autonomy, distribution, and consistency.

The *service* requirement is to provide continuous service of as much as the system as possible in the face of failures, including node and network link failures. Local programs should be localized, with replication of data and processing allowed for throughput or backup purposes. Replication can increase the availability of a service and allow graceful degradation in response to failures.

The usual concept of *extensibility*, where hardware may be added to a system in order to increase processing speed, allow additional on-line data storage, etc., is generalized to allow the addition of entire nodes to the network. These additions may be physical, where new communication links make additional nodes reachable, or they may simply be logical, where an application system expands its processing and data across additional nodes in order to increase reliability, processing or data capacity, or response time. Both physical and logical changes must be dynamic, so that application systems need not be shut down and restarted in order to utilize the changed environment.

Since the individual nodes may be owned and/or controlled by individual organizations, each node must be allowed a high degree of *autonomy*. The owner of each node must be able to restrict access to the data and services available at that node. A distributed application must not be able to override restrictions present at any node that the application uses.

The fact that autonomy may be enforced at specific nodes makes ARGUS more suitable to the client/server [84] model of distributed processing. In this model a job need not control the placement of its distributed sections. Rather, server processes are distributed and may be contacted as necessary. The server process responds to requests, but is not controlled by them. In this way these server processes may be controlled

locally at their physical nodes.

A distributed application allows true *concurrency*, where two or more processors are simultaneously executing separate threads of control within a single application. Concurrency optimizes throughput and response time. It is a major factor in the ability of networks of relatively low power processors to replace much more powerful (and expensive) mainframes for those applications which can be efficiently distributed.

Any system in which data is being read and modified must emphasize the maintanence of *consistency*. To guarantee consistency becomes more difficult when the processes performing the data modifications are distributed. The concept of locks must be generalized and expanded to handle slow and relatively unreliable communication links. Efficiency is inversely proportional to the length of time a lock is held and the time will in general be much longer in a loosely coupled system, as compared to a uniprocessor or tightly coupled multiprocessor system. Data replication adds another layer of complication; not only must a set of data be consistent but it must be consistent with its replicated copies.

The above requirements are used as the basis for the design of ARGUS. The purpose of ARGUS is to allow the high level language programmer to design and implement a distributed program meeting these requirements through facilities internal to the language environment. The actual syntax of ARGUS is based on the object oriented language CLU, which supports a wide set of abstraction mechanisms. These support well structured programs, a prerequisite for separation into distributable modules. ARGUS extends CLU by defining additional abstraction mechanisms.

The ARGUS approach to maintaining consistency in a distributed environment is based on the *atomic activity*. In ARGUS, an *activity* can be thought of as an operation that examines and then changes the system state from an initial value to a final value, with possibly intermediate values. An ARGUS activity is *atomic* if two properties are present: indivisibility and recoverability. An activity is *indivisible* if the execution of the activity appears to neither overlap nor contain the execution of any other activity. This means that the final values of any system state changes performed by an atomic activity depend solely on the initial system state at the commencement of the activity. An activity is *recoverable* if the effect of the activity on the final system state is all-or-nothing. That is, if the activity does not complete successfully then the final values of any system state changes performed by the activity must be maintained at, or restored to, their initial values.

Atomic activities therefore see only the final states of values affected by other atomic activities. The atomic activity requires an overhead as the cost of maintaining consistency, but if it were not built into ARGUS an equivalent mechanism would be required anyway. Atomic actions are central to ARGUS and are called *actions*. Values examined and modified by actions are called *objects*. An object may be as simple as a basic data item, or as complicated as our usual concept of an object as encompassing any amount of data along with operations to manipulate that data.

An object is *stable* if it is maintained on a stable storage device. An object is *atomic* if it is protected by synchronization and recovery management. Since the overhead is high, objects are defined as atomic only when necessary. For example, an object referenced only by a single activity

could be defined as non-atomic. An activity is an atomic activity (ie: an action) that requires that all shared objects referenced by the action are atomic objects.

ARGUS implements atomic objects through a basic locking mechanisms. There are only two types of locks: read locks and write locks. The mechanism is standard: multiple readers are allowed but a write lock allows no other readers nor writers. When a write lock is granted a *version* of the object is created. This version is the copy of the object acted upon. If the activity completes sucessfully, then the updated version replaces the original object. Otherwise, the version is discarded. In either case, the write lock is then released. An activity that completes successfully is said to *commit*, an unsuccessful activity *aborts*. An activity may obtain multiple write locks, but there is always a single *commit* to make object changes permanent and release the locks.

A failure at a node which cannot be handled by the ARGUS system, such as a harware failure, causes the ARGUS system at that node to crash. Activities with write locks on objects at that node are aborted, since possible intermediate values in the current version are no longer available. ARGUS attempts to implement paths to replicated copies of lost objects. This node failure procedure is actually simply a collection of abort steps which ARGUS would implement individually in the case of a single object failure.

Recall that an action is an atomic activity that must commit before changes it makes to objects become permanent. While an action must be atomic when viewed externally, its internal structure should allow for modularization to promote structuring. For this, ARGUS allows (and encourages) nesting of actions. Sub-actions may be performed

sequentially or cuncurrently. Sub-actions appear to be atomic to each other, following the same commit or abort procedure as all actions do. However, the commit of a sub-action is conditional. It appears to the other sub-actions (at the same nesting level) that the commit is permanent, but the parent action continues to maintain the updated object as a version. Only when the highest level action commits do all objects affected by it and its descendents become updated with the temporary versions. At this point, the write locks are released.

An action has total responsibility for its sub-actions. It can at any time commit or abort. In either case, this aborts all sub-actions still in progress. This provides a convenient aprroach to such uses as "commit when a majority of sub-actions have committed" (consensus) and "immediately abort if a single sub-action has aborted" (stop on error). An action cannot execute concurrently with its sub-actions. It defines the subactions, their sequential and concurrent ordering, and the procedures to follow as the result of sequences of sub-action commits and/or aborts. After the sub-actions have commenced, the parent action cannot regain control until all sub-actions have terminated. An early termination, such as a time-out, must be predefined at the sub-action start point. For example, a timer subaction can be started concurrently with the processing sub-actions, and a commit by the timer action could be defined so as to force aborts by any remaining processing sub-actions.

The single writer, multiple reader, lock mechanism must be extended to handle nested actions. An action may obtain a read lock on an object despite the presence of write locks if all write locks are owned by an ancestor action. An action may obtain a write lock on an object despite the presence of read and/or write locks if all such locks are owned by an

ancestor action. Recall that actions cannot execute concurrently with their sub-actions. Since the extended lock rules allow for a non-exclusive write lock only if all other locks are held by an ancestor action, and since those ancestor actions cannot be executing concurrently with the present writer, it is sufficient to maintain a stack of versions for the object. A commit causes the top version in the stack to become the current version for the parent action, while an abort causes the top version in the stack to be discarded.

Remote procedure calls, and messages in general, follow *at-most-once* semantics. That is, a message is delivered exactly once, with a reply received, or the sender is so informed and allowed to choose the appropriate action. This allows communications to be treated as sub-actions. A failed communication becomes a sub-action which aborts. So, the powerful ARGUS sub-action sequencing and termination controls are available for remote communication.

Actions (atomic activities) and atomic objects are the basic ARGUS building blocks. A high level view would show that an ARGUS distributed program consists of a group of *guardians*. Each guardian encapsulates one or more resources and controls access to these resources. Each guardian provides as an external interface a set of operations, called *handlers*, which can be called by other guardians through remote procedure calls. Internally, each guardian contains data objects and processes; each data object is either global to the guardian or local to one specific process within the guardian. Each guardian exists entirely at a single physical node.

ARGUS provides support for restarts of guardians after node crashes. This assumes the availability of the stable objects, possibly

through alternate paths to on-line storage devices or through replication. The language support system recreates the guardian using the stable objects. Each of these objects is current as of the last commit of a high-level action. The guardian must provide a predefined process to recreate any non-stable objects. An example of a non-stable object might be a running total which can easily be recalculated from some set of stable objects, so that it is not worth the overhead involved if the running total had been declared as a stable object. Versions of objects will have been lost, but then that is what would be expected since a node crash can be thought of as causing an abort of all current actions.

Guardians and handlers can be thought of as an abstraction of the physical distributed system, with the handlers providing for communications between the independent guardians. Several guardians may be present at a single physical node, but inter-guardian communication is independent of location. The at-most-once semantics of messages (including handler calls) ensure that the handler either succeeds completely or has no effect. The ability to form communications as sub-actions allows for concurrency while awaiting a response. A call to a handler causes the ARGUS executive to form it as a message and forward it to the appropriate guardian. However, the sender can treat the call as simply the execution of an action since the eventual return message will be presented by ARGUS as a commit or an abort.

The structure of actions, atomic objects, locking mechanisms, and concurrency restrictions of actions and sub-actions, together provide a high degree of synchronization for concurrency. However, there are some additional considerations. For example, guardians may include processes to run in the background and processes to handle certain recovery

procedures. These run outside the action system. In addition, the user may occasionally want to employ a locking structure more general than that provided, such as to allow simultaneous read access to an action and write access to a concurrently executing action. (ie: rapid response may be more critical than absolutely current data.) For this, ARGUS provides an object called a *mutex*. Each mutex controls a critical region and in addition has a data object associated with it. While any process is executing in the critical region, no other process is allowed to execute in that region and the data object cannot be written to stable storage. So, even though other actions may have a write lock on the data object, any commit is delayed until the process currently executing in the critical region has left the region. This can be used to guarantee that only consistent states of the data object are ever written to stable storage.

The ARGUS language provides a high level approach to the distribution of processing. It guards the distributed data by synchronizing concurrent access to the data, by requiring that all calls to a guardian be approved by a handler within that guardian, and by providing stable copies of the data so that the guardian can be restarted after a node crash.

Despite the sub-action restrictions on concurrency, the system must do quite a bit of work to assure synchronization. For each atomic object, a history of actions must be maintained. As sub-actions commit and abort this history is updated. When the top-level action commits, the history must be consulted to assure that any other guardian which called us through a handler that touched this particular data object has not since aborted. Only after all this can the object be updated on stable storage.

The concept of ARGUS includes significant considerations for fault tolerance. Data rollbacks are inherant at several levels. Inter-guardian communication is location dependent and guardian restart in the case of a node crash is supported. It must be noted that there is no consensus mechanism. That is, any failure is assumed to be detectable immediately, with no allowance for Byzantine behavior by any part of the network.

2.4.6.3 *MOD

*MOD (pronounced 'starmod') [19] is a high-level language designed to facilitate the distribution of tasks within a program across a loosely connected network of processors. Language constructs are provided which aid in the development of programs utilizing multi-tasking. The language syntax which supports this multi-tasking over loosely coupled systems is very similar to general purpose language syntax to support multi-tasking on a single system.

A *concurrent program* requires multiple processes for its implementation. A *distributed program* can be characterized as a concurrent program which requires multiple processors amongst which its multiple processes can be allocated. *MOD is intended to generate distributed programs for both applications and systems functions. Because it must support systems functions, *MOD is designed to be transparent. A language is *transparent* if any system state which could be obtained by programming the component machines could also be obtained by using the language. In particular, *MOD allows the programmer to define and schedule processes, to allow processes to change in response to workload and response time requirements, and to orgainize the program to reflect the structure of the network and the capabilities of the individual processors.

As might be guessed from its name, *MOD is based on the *module* concept of the Modula language. A *module* consists of the interface specification, data structures, procedures, processes, and initialization information. *MOD extends the module types and options to allow for interprocessor communications.

The highest level *MOD module, which defines the program itself, is called a *network module*. This module defines the structure of the network and the links between the individual *MOD processors. Note that *MOD uses the term *processor* to define that part of a program which will run on a single loosely connected system. For example, if one of the loosely connected systems was itself a tightly coupled dual processor, then that portion of a *MOD program (network module) assigned to the multiprocessor could use both physical processors but would still be considered a single *MOD processor.

While the main purpose of the *MOD network module is to detail how to assign the *MOD processors amongst the physical processors and to define the intercommunication links, a network module may also define constants, procedures, and processes which will be global among the *MOD processors. The compiler simply copies them into the individual processors; their presence in the network module assures consistency.

The *MOD processor module contains the *MOD processes which can be multitasked, along with global data and procedures that can be shared among these processes. All data and procedures can be local to a single process or procedure, or can be globally available either read/write or read only. Global definitions use import and export attributes. When a module (process or procedure) is imported, all of its internal names with the export attribute are available to the importer. To minimize import

lists, the *define* statement may be used for a name rather than the export attribute. In this case the name does not appear in the import list but the name may be referenced by the importer if the name is prefixed by the imported module name. The purpose of all this is to allow the programmer to be able to micro-manage the global scope of each individual name.

The central *MOD communication mechanism is the message. A message may contain no data (in which case it acts as a signal), or it may contain an arbitrary amount of data. Messages are structured, with type checking enforced at both the sending and receiving end. Messages can be used for *any* type of communication among processors, processes, and/or procedures, within a *MOD program. Processes and/or procedures within a single processor can also communicate through shared variables. The advantage of using messages rather than shared variables when communicating between processes is that a process can then be moved between processors with little reprogramming required.

The *MOD definitions for processes and procedures are very similar. This allows the programmer to easily convert from one to another (recall that each is activated by the receipt of a message) to optimize its use. A procedure call involves less overhead (a frame on the stack) while a process call allows concurrency. And, processes maintain individual dynamic priority levels. Several instances of a single process may be executing concurrently.

A processor can handle incoming messages by defining a message handler process. It can also define message ports. A *port* can be thought of as a message queue from which messages may be scanned and/or extracted when desired. The sender is unaware of how messages are handled at the receiver. The sender need not even be aware of whether the

receiver is a port, a process, or a procedure. The only restriction is that the receiver of an interprocessor message may not be a procedure, it must be a port or a process. This is because if it were a procedure it would be necessary to interrupt a process and activate the procedure asynchronously. The process would then be required to wait for procedure completion. With a port or process handling the communication, the priority scheme between processes would be used and the interrupted process could resume based either on priority or on a wait (such as for I/O) by the message handler process. A port simply queues the message for later processing.

*MOD has been installed both on bare machines (PDP-11's) and under UNIX. The advantage of running under an operating system, besides sharing the machine with non-*MOD users, is that a *MOD distributed program may be tested with multiple *MOD processors on a single physical processor, and dynamic message routing may be added at a level lower than the *MOD kernel, since *MOD itself requires that message routing be user defined within each *MOD program.

*MOD aims to allow the programmer to structure a program so that parts of the program can be distributed over a loosely coupled network. It does not attempt to make the distribution invisible to the programmer; rather it enforces structures that simplify the distribution. But, the programmer must understand the network structure, including the capabilities of the individual systems and the intercommunication links, and must structure the *MOD program accordingly. The programmer is expected to take advantage of the ease of module redistribution to tune and optimize the program, and to retune and reoptimize to meet changing workloads and hardware configurations.

2.4.6.4 Synchronizing Resources (SR)

SR [4, 5] is a language for programming general purpose distributed systems. It supports a wide variety of distributed support mechanisms, including local and remote procedure call, rendezvous, dynamic process creation, synchronous and asynchronous message passing, multicast, and semaphores. This flexibility is provided through a small number of integrated concepts, which is intended to make the language easy to learn and allow for efficient implementation.

The main language constructs are *resources* and *operations*. Resources encapsulate processes and variables that they may share, while operations provide the main mechanisms for process interaction. The design goals of SR are ease of use, efficiency, and *expressiveness*. Expressiveness means that the language should allow the programmer to solve a distributed programming problem in a straightforward manner. In general, a distributed program (as compared to a sequential program) will be more likely to shrink and grow dynamically in response to current activity. A distributed program will always employ multiple threads of control. Each thread is the equivalent of a sequential program. Thus, a distributed programming language should provide more control mechanisms than a sequential programming language.

SR attempts to generalize concepts that are familiar from sequential programming languages. Resources generalize modules, and operations generalize procedures. An SR program is composed of a number of separately compiled *components*. There are three kinds of components: resource specifications, resource bodies, and globals. Resources are the main building block – they contain processes and data shared by the encapsulated processes. Globals contain declarations of constants and

types shared by resources (separate resources may not directly share data).

A resource is actually a parameterized pattern. Instances are created dynamically as needed. Each resource defines operations between the processes within the resource. These processes may share variables. Each resource is composed of a specification and a body. The specification declares external components to be imported from other resources and internal components to be exported to other resources. The body contains the processes of the resource, declarations of objects shared by these processes (but not exported to other resources), and initialization and finalization code.

The body of a resource inherits all objects declared (or imported) by the corresponding specification. The body is parameterized; actual parameters are employed when an instance of the resource is created. Processes within a resource may contain import requests that are in addition to the import requests in the corresponding specification. This is to avoid the overhead of large import lists which include infrequently used objects. The disadvantage is that a resource specification cannot be used to describe the entire external range of the resource.

A resource instance is dynamically supplied as the result of the *create* statement. Arguments are passed to the newly created instance, and the initialization code is entered. The create statement blocks only until the initialization code of the created instance completes, after that the creator and the new resource instance run concurrently (on the same or separate processors).

The creator resource may execute a *destroy* statement to terminate the created instance. In this case, the finalization code of the created instance is entered. The destroy blocks until the finalization code has completed and the space allocated to the new instance has been freed.

The creator is given full control over the created instance. By default, the new instance is created on the same processor as the creator unless the creator specifies an alternate machine.

Just as resources are patterns for objects (instances of the resource), operations are patterns for actions on objects. Operations are declared in *op* declarations. These can appear in resource specifications, resource bodies, or within individual processes. Operations are invoked by *call* or *send* statements. The target process services these invocations through *proc* (procedure-like) or *in* (input-like) statements. Arguments may be passed by value, result, or value/result. They may also be passed by reference, but only within a single address space.

A call statement terminates when the service has returned results or terminated, possibly abnormally. A send statement terminates as soon as the parameters have been successfully stored on the target machine. So, call is synchronous (like a standard sequential procedure call) while send is asynchronous. Technically, send is semi-synchronous. This is because the sender is assured that the receiver is accessible and his message is available to the receiver, even though the sender continues with no assurance that the message will actually ever be serviced.

An operation may be invoked by a call or a send (unless the operation itself enforces a restriction). As described for the create statement, the invoker is given as much control as possible. Of course, if the invoker

uses a send and is expecting return data, it must have provided for asynchronous communication from the servicer or must periodically check for results.

An operation may be serviced by a proc statement or by one or more in statements. A proc is a generalizaion of a procedure declaration, although it will actually be invoked as a process. If invoked by a send (rather than a call) it may therefore execute concurrently with its caller. An in statement is a list of operation commands. Each operation command is similar to a proc statement, including a parameter list, but each operation command includes a synchronization and scheduling boolean expression. The scope of these expressions is very powerful. The may use current invoker paramenters, system status information, precedences amongst themselves, etc. An invoker serviced by an in statement is blocked until at least one of the operation commands is allowed to be invoked (because its synchronization and scheduling expression evaluates to *true*).

It can now be seen that various synchronizing mechanism are available through these few statements:

call/proc	procedure call
send/proc	process forking
call/in	rendezvous, semaphores
send/in	asynchronous message passing

Several other communication primitives are supplied to promote efficient distributed communication. For invoked routines, the *return* statement operates as expected. The use of a *reply* statement appears to the caller to be a return (the caller will recieve return data and become unblocked) but the invoked routine will not terminate. This facilitates

conversations between processes. Another primitive, the *co* statement, supports concurrent invocations. The co statement contains a list of <invocation, post-processing> pairs. The execution of the co statements results in the concurrent start of all invocations. As each invoked routine completes, the corresponding post-processing code is executed (sequentially, by the caller). Each post-processing block may terminate normally or may execute an *exit* statement. The co statement terminates immediately upon execution of an exit statement by any post-processing block. Otherwise, the co terminates when all post-processing blocks have terminated. A terminating co statement does *not* terminate any still executing invocations. They are allowed to continue concurrently until they terminate, although the post-processing code will not be executed. This is useful in consensus operations or in situations such as multiply replicated data where an acceptable number of replications have been successfully performed, but additional replications are allowed to complete even though successful completion is not critical.

SR supports two mechanisms for distributed failure handling: invocation handlers and remote monitors. An invocation handler is declared with the resource control statements, or as part of the proc declaration. The handler is run if an invocation error is detected by the run time SR system, or if the invoked operation executes an *abort* statement. This handler could, for example, return a status code to its invoker. Rather than count on the invoked routine, an invoker can make use of the *when* statement to monitor failures. An invoker who uses the send invocation is unblocked as soon as his message is delivered to the machine of the receiver; this doesn't mean that the invoked process will ever successfully process the message. The when statement request that the SR run-time support monitor the when parameter, which may be a physical machine,

virtual machine, resource instance, or individual process. An asynchronous failure of a (possibly remote) object results in an asynchronous call by the SR run-time monitor to our routine declared in the when statement.

SR has been implemented in a UNIX version and a stand-alone version is under development (at the University of Arizona). The UNIX version consists of a compiler, linker, and run-time support (RTS). The compiler produces C code, which is then passed through the local C compiler to produce machine code (MC). The linker takes as input previously compiled resource specifications and bodies, and globals, along with a specification list of the physical machines on which the program is to execute. One instance of a *main* resource is defined, along with a main physical machine.

Upon program invocation, the main resource is started in a UNIX virtual address space (VM) on the main machine, which starts the RTS on each physical machine. The main process then spawns other processes on other VMs, local and remote. Concurrency within a VM (address space) is not supported directly by UNIX, so it must be simulated by SR run-time support modules in each VM. UNIX sockets are used for inter VM (and inter-processor) communication. The RTS hides the details of the network environment from the MC.

Notwithstanding the above descriptions, the UNIX SR environment optimizes for efficiency as much as possible. For example, a call to a service within an address space is serviced by a standard on-the-stack procedure call, rather than by a separately invoked process. Failure monitoring can be simplified within a single machine. Processes are not dynamically relocatable by the RTS. Therefore, anytime communicating processes

are on the same VM or physical machine, efficient local system level mechanisms may be substituted for the more general distributed equivalent. These are hidden from the programmer.

SR implements a more general set of facilities for distributed control than does *MOD. However, the SR programmer is faced with the need to synchronize the effects of his *resources* to a greater degree than the *MOD programmer must with his *processors*. There is one communication mechanism in *MOD and the caller is totally unaware of whether the invoked service is a procedure, a separate process, or even a separate processor. SR allows for a wide variety of synchronizing mechanisms, but the invoker and invoked service must coordinated their intercommunication. For both languages, the programmer is given complete control over the distribution of resources around the network, but the network is treated as static during run-time. Fault handling is limited to detection and reporting; fault tolerance is the responsibility of the programmer.

Extensions to SR have been proposed [72] to aid in the handing of asynchronous failures. The aim is to treat these failures as just another class of events and to handle them as instances of any other class of normal system events. A new type of variable called *binding* is defined, which in effect is bound to information from and about a remote object. The binding variable is maintained by the RTS and can be efficiently inspected (or monitored) locally. The binding variables can be used for synchronous or asynchronous response to a variety of failure events.

2.5 Fault Tolerant Networks

The main purpose of the distributed hardware and software

systems discussed thus far has been to distribute processing and data so as to increase total throughput of distributed jobs. Fault tolerance has basically appeared as natural features of the loosely coupled environment, such as allowing remote replicated data and static or dynamic relocation of processing elements, usually at the direction of some central authority within each application job.

Any network utilizing such features can be called fault tolerant. In some cases, however, fault tolerance is central to the network design rather than an additional feature of a network designed for throughput.

2.5.1 Configuration Reliability

Even assuming that individual reliabilities for nodes and links within a network are known, simply determining the reliability of communications between any two nodes is difficult. In fact, even in the simplified case where only links can fail (each with some given probability) and the failures are independent, determining the probability that any pair of nodes can communicate has been shown to be NP-hard [28]. Nevertheless, numerous algorithms have been proposed for computing network reliability which operate efficiently on reasonable network configurations.

One such technique [1] is to take the graph representing the network and from it construct an equivalent tree. This tree will in general be much larger than the original graph. The algorithm, which calculates reliability measures as the tree is being constructed, is, of course, still NP-hard. But, in practice it is useful for moderate sized networks. And, improvements [2] have been developed for the algorithm which make "better" next branch expansion choices which decrease the average run time. Von Neumann [90] studied the effect of random failures on

formulae. A formula is a circuit in which the inputs to each gate are independent and the output of each gate is the input to at most one gate. Both of these restrictions are violated by most networks. This work was therefore generalized at IBM Almaden Research Center [25]. Two features of the work by Von Neumann were shown to still hold:

1. The error probability for the entire network approaches $1/2$ as the failure probability for each gate approaches a limit strictly smaller than $1/2$.
2. The *depth* of a network must increase as the network reliability increases in the face of individual gate failures. Depth is the number of gates on a longest path between input and function output.

These features seem reasonable for both formulae and networks but the details of the relationships between individual gate reliability and network reliability are quite distinct.

The concept of *fault distance* [39] is used in diagnosability and distinguishability analysis. Fault distance is a measure of the number of simultaneous failures which can be *diagnosed* (a failure is detected) or *distinguished* (the specific failure is identified).

If the nodes are physically located sufficiently close together a switching network can be employed to dynamically control interconnections. Various forms of crosspoint switching networks [89] can offer various degrees of fault tolerance through replicated busses and individually switchable crosspoint connections.

The simple act of broadcasting to a group of distributed processes entails a trade-off between minimization of process blocking and reliability of message transfer. Birman and Joseph [12] define a number of

communication primitives to implement various levels of communication reliability.

2.5.2 Routing and the ARPANET

Routing in a network can be *static* or *dynamic*. Static routing implies that a source node already knows the optimal route to the destination node. This route is sent along with the message, so that intermediate nodes simply follow the message routing instructions. Each node in a static routing environment must know the optimal route to all possible destination nodes. There may be a central routing authority which knows the network topology, calculates optimal paths, and sends the appropriate data to each node. Or, each node may know the network topology and calculate its own optimal path. Dynamic (or *adaptive*) routing does not require that each node contain a complete routing table. Instead, whatever information is known about the destination is used to choose the next node in the path. This node repeats the best next node choice, and so on until the destination is eventually reached.

Static routing algorithms have an obvious advantage since the sending node knows the next node in the optimal path. In the case of large networks, this is clearly unreasonable. However, the effects of link and node faults can be more easily studied and the results are applicable to networks with dynamic routing. Dolev [23] uses the static routing environment to characterize the effect of failures on various network configurations. The *distance* of a network is defined as follows: Pick some measure of message transmission efficiency between two nodes. This could be hop count, total delay assuming average line usage, etc. For each pair of nodes, find the best path under this measure. The maximum of these measure calculations is the distance. For example, under hop count the

distance would be the maximum number of intermediate nodes to send a message from a source node to a destination node along the shortest path. By characterizing the effect on network distance of one or more failures, we can calculate the eficiency of various network configurations during periods of failure.

Most networks, and especially large networks, use dynamic routing. While not guaranteed to produce the optimal path, dynamic routing allows each intermediate node to use its current knowledge of line loading, inter-node delay times, down nodes and lines, along with partial network topology, to choose what it considers the optimal next route segment. It is unrealistic for nodes in a large network to maintain all this data about the entire network. And, even if this were possible there would be significant timing delays so that all nodes could not simultaneously hold identical views.

The original ARPANET algorithm [18, 54] is a distributed dynamic algorithm. It in fact assumes that the network is small enough that each node can determine the optimal path to the destination node. However, the algorithm recognizes inconsistencies due to timing delays in status broadcasts. So, even though a source node thinks it knows enough to pick the path with the smallest total transit time, it does not predefine the entire path. As the message reaches each intermediate node the optimal path toward the destination is recalculated based on the network status data currently available to that intermediate node. Each node maintains minimum delay tables to all other nodes. Each $^2/_3$ of a second, a node broadcasts its updated tables to each of its neighbors. This means that each node receives a minimum delay table from each of its neighbors every $^2/_3$ of a second and uses this data plus its calculated delays to its

neighbors in order to update its own delay tables.

As the number of nodes in the ARPANET increased, the size and frequency of delay table transmissions caused the original routing algorithm to become unwieldy. Inconsistencies between delay tables in various nodes became more pronounced, leading to increasing instances of anomolous behavior. The replacement routing algorithm [55] is actually very similar to the original. One difference is the manner in which delay tables are propagated. Each node transmits delay information only about itself and its neighbors. This information is broadcast to all other nodes. Each node therefore receives these delay table updates from all other nodes rather than just from its neighbors. But, these messages are much shorter than the corresponding messages in the original algorithm. And, each message contains only delays which have changed by some defined increment since the last transmission of that particular delay. So, small delay oscillations around some average value are ignored. Each node maintains its best path to destinations data in a tree structure, rather than a table. Dijkstra's shortest path first (SPF) algorithm is used to generate, search, and update the tree. This replacement algorithm retains the original philosophy: each node should determine the best path to each destination, but intermediate nodes recalculate the best path based on their current knowledge.

2.5.3 Autonomous Decentralized Systems

The above description of the ARPANET routing algorithms shows that the ARPANET nodes under the ARPANET routing algorithm form an autonomous decentralized system. While we generally think of autonomous systems as systems where human intervention is impossible during system operation, such as on space missions, the ARPANET routing

routines do in fact route around failed nodes and send required start up and join information to nodes returning on-line. However, the ARPANET is not a general purpose computer system.

The following discussion centers on autonomous decentralized systems which are capable of general purpose computing and are physically close enough together to allow arbitrary network configurations. Also important is the concept of limited repair capability. The reliability of systems with limited repair must take into consideration the existence of a largest continouous operational time before a complete system failure [31]. For an automomous, non-repairable system, the probability distribution of this operational time to complete system failure is a primary factor when evaluating different system configurations.

2.5.3.1 Planar-2 Network Configuration

An autonomous decentralized system performs both processing and control functions in a decentralized manner [37]. Since there is no master node, the emphasis is on configuring the network so that failures cause the least disruption possible between the non-failing components. Typically, these configurations include many independent links between the nodes.

In a planar-2 network configuration, the nodes are logically connected in a 2 dimensional plane, with horizontal and vertical links. Figure 10 shows a 16 node example. In this case, there are 8 links, with each link connecting 4 nodes. Each of these links could actually be installed as three separate links. In this case, nodes would require up to 4 physical ports each. The system would become more fault tolerant since each link would now connect only 2 nodes so a failed link would cause a lower

overall system degridation. But, beside the extra cost and complexity of the additional links and node physical ports, the distance of the network (with respect to hop count) would increase from 2 to 6

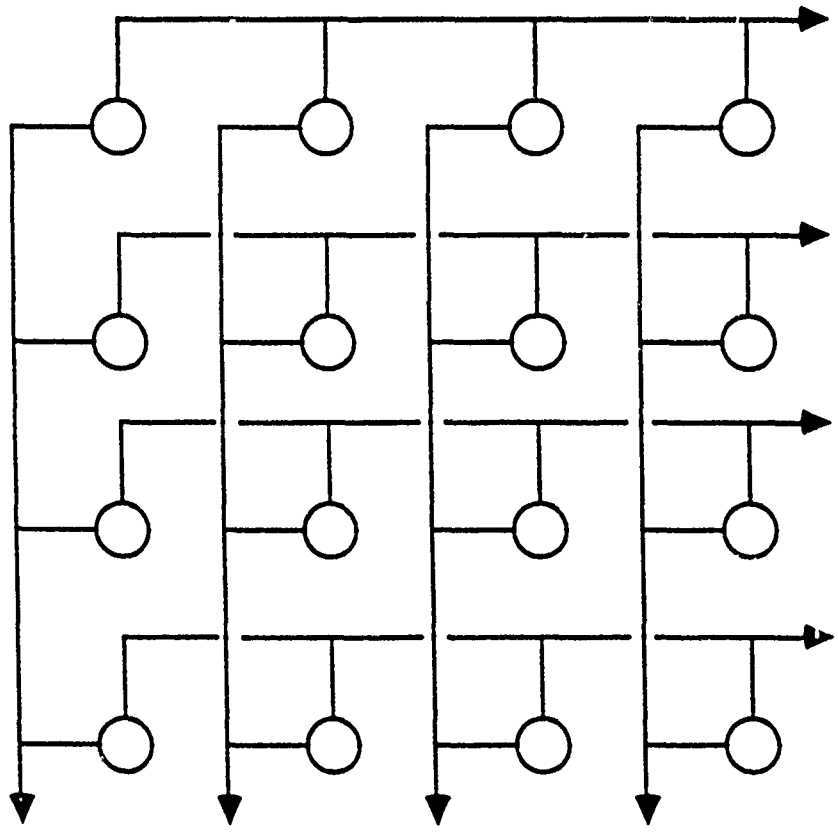

Figure 10. Planar-2 Network Topology

2.5.3.2 Advanced On-Board Signal Processor (AOSP)

Sensor satellites were initially designed to collect data and pass it unprocessed down to earth based receiving stations. As the data gathering ability of satellites steadily increased, so did the necessary bandwidth, to the point where it was becoming a limiting point. The solution is to perform the data reduction on board, thereby requiring the transmission to earth of a much reduced volume of data. This in turn requires an increase

in the computational power of the on-board computers, and with this increase comes a decrease in reliability of the now more complicated on-board systems. But, the limited space and load carrying ability of the sensor satellite limits the options available for fault-tolerance.

The AOSP project [70] had as its goal to find a general solution to this problems. That is, while allowing flexibility of choosing the specific system for a specific sensor processing environment a general design to address this problem was needed. It turns out that sensor processing algorithms are generally easy to break into discrete steps. So, a network approach, where one processor does some reduction and then passes the intermediate result as a message to the next processor, turns out to work very well in practice. The challenge then became to design a network structure and the associated control program support for acceptable levels of fault tolerance to allow for planned five to ten year missions while allowing flexibility in choices of processors and online storage.

The two main areas of research were the network structure and the distribution of functionality of the software support among the nodes in the network. Ideally, all processors in the network would be identical, so that the spares could take the place of any failing processor. The design of sensor systems allows for dropouts and short periods of lost data. This vastly simplifies the start-up of a backup system – no attempt need be made to recreate the lost data. So, in case of a processor failure it is necessary only to start up the failing tasks in other similar processors and then begin to redirect messages.

An early design decision was to "split" the processor at each node [66]. That is, at each node there will be a standard Node Control Unit (NCU) attached to the network and also an Application Processing Unit

(APU) to run the sensor processing programs (see Figure 11). As previously indicated, it would be nice if the processors chosen to be the APUs for a particular AOSP system were identical, but this is only to make it easier to provide spares. If the APU processors are not identical then it will be necessary to provide several types of spares. In any case, the APU type(s) can be chosen as appropriate for the specific sensor processing desired. The NCUs have the sole purpose of running the distributed Network Operating System whose tasks are:

1. know the physical node at which each active task is currently operating so as to be able to route messages correctly from application to application.
2) know the primary and backup locations for each task so that all tasks from a failed processor may be restarted.

Each node, of course, has its own clocks and power supplies so that even though all processors must of necessity be placed in close proximity to each other the network is in fact loosely coupled. The concept of sparing in a loosely coupled network allows most of the processors to be performing useful work (as opposed to the triple modular redundant design) and we even have the option of using all processors for maximum efficiency and then in case of a failure choosing the path of degradation. For example, we could run more tasks/processor (possibly losing some data during high throughput periods) or we could stop some tasks thereby omitting certain forms of processed data or we could even periodically switch between tasks.

The other important network related decision is in regard to the interconnection scheme. Multiple networks are desireable both for the fault tolerant aspect and because of the expected large volume of message

traffic. While a fully connected network (all point-to-point connections possible. are made – this of course requires $N*(N-1)$ network links) has theoretical niceties (e.g., message forwarding is never necessary regardless of the number of node failures) there are practical limitations both on the number of links and the number of connections a single NCU should be expected to be able to respond to.

The final decision was a compromise on what is called a planar-4 connection scheme [65]. Recall that in a planar-2 connection scheme the N nodes are arranged in an $\sqrt{N} x \sqrt{N}$ array. Each node has two network connections, one to a horizontal link and one to a vertical link. In planar-4 the nodes are similarly arranged, but each has four network connections. These are the usual horizontal and vertical links, and in addition a left ascending link and a right ascending link. The total number of network busses is $4\sqrt{N}$ and there are $\sqrt{N}$ nodes on each bus. The diagram for a 16 node planar-4 network is shown in Figure 11.

The NCU at each node runs a copy of the Network Operating System (NOS), which provides communication between the NCU and its attached APU along with inter-node communication to the other NCUs (at the other nodes.) The Global Operating System (GOS), which contains Health Managers (HMs) and other facilities for testing the status of each node, is distributed among the nodes [67] in such a way that multiple copies of critical sections are present. As can be seen in the above diagram, the worst possible single point failure – a node goes down and brings all 4 attached network links down with it – leaves 12 network links to handle the remaining 15 nodes and there will remain a distance-2 store-and-forward link (ie: one intermediate node) between any of the remaining nodes. As network size expands beyond the above 4 x 4

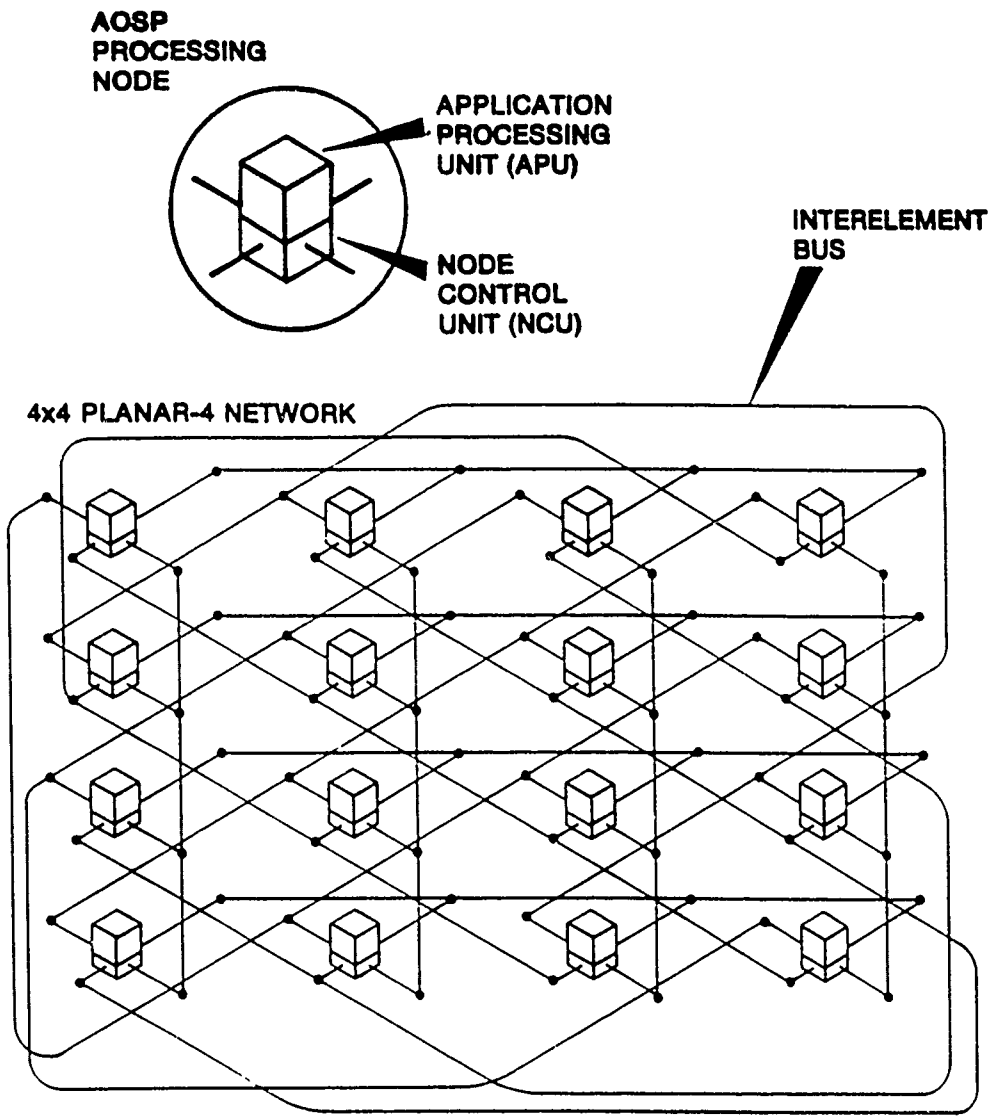

Figure 11. AOSP APU/NCU Planar-4 Connection Scheme [65]

example, the large number of distance-2 store-and-forward links helps maintain short forwarding link sequences.

In conclusion, the AOSP is a design for a highly fault tolerant (in terms of the signal processing environment) computer system network with minimal processor redundancy. The network bus redundancy is considerable, but is limited to four connections per node control unit for any

size network. The system is very flexible (any size network is supported and the splitting of each node into a node control unit and application processing unit allows the choice of the appropriate APU for the job) and can be adapted to many environments. The limitation is that it was designed for an environment where data dropouts are expected and tolerated, which greatly simplifies the restart of tasks from failing processors to spare processors in the network.

CHAPTER III

FTM DESIGN PHILOSOPHY

3.1 Introduction

The aim of this research is to formulate protocols which provide fault tolerance services to applications in an otherwise familiar general purpose hardware and software environment. As the literature survey demonstrates, hardware replication is an integral component of any fault tolerant system. For special purpose systems where reliability constraints greatly override hardware costs, standby or shadow systems are appropriate. To combine fault tolerance with high system utilization the approach is to fully utilize the hardware and, as components fail or otherwise become unavailable, to redistribute and reprioritize the workload among the remaining components.

Tightly coupled multiprocessors and loosely coupled networks can each be utilized in this fashion. Multiprocessors allow for more efficient distribution of resources (processors, memory, I/O devices) among active tasks, along with much more rapid inter-task communication. Synchronization is simplified (since system clocks are directly interconnected and thereby can be exactly synchronized) and therefore checkpointing can be more easily implemented. This means that recovery from many types of error can be, and often are, automatically invoked by the operating system in a manner totally invisible to the application tasks.

While a loosely coupled network cannot dynamically distribute resources among tasks, it can distribute tasks among resources. This is not nearly as efficient as the distribution within a multiprocessor. To dynamically redistribute tasks within a network is much less efficient and

more difficult. However, networks have many other features. One of these is the current sub-linear cost versus computing power curve. That is, a mainframe costs more per unit of computing power than a less powerful machine. So, even ignoring the network feature of geographic dispersion there is a current trend toward the use of local area networks where until recently a mainframe would have been appropriate.

3.2 Hardware Environment

The target hardware environment for this research will be a loosely coupled network of general purpose computer systems. The surveyed fault tolerant networks implement high levels of fault tolerance through specific hardare design and interconnection schemes. For example MEDUSA, a semi-loosely coupled network, uses a single address space for all processors. Special hardware node interconnect devices hide the lack of locality of reference, and a distributed operating system optimizes the locality among communicating tasks.

The network node interconnection scheme can be designed to provide high levels of node to node paths. The AOSP project in particular defined a very high redundancy of node interconnections. The emphasis on fault tolerance in AOSP was extended to the point that each node was devoted totally to activities which support fault tolerance, such as message forwarding, distributed monitoring of nodes for correct operation, and relocation of application tasks in the face of failures. A separate processor was attached to each node to accomplish the application processing!

3.3 Software Environment

Distributed languages provide the application programmer a tool to support fault tolerance in a standard distributed network environment. A

higher level of support is offered by distributed operating systems, so that the user is not restricted to a specific language. LOCUS, a prime example of this approach emulates the UNIX user interface. This allows the programmer to utilize a variety of computer languages in a universally familiar software environment.

3.4 Layered Fault Tolerance

This research investigates the possibility of offering the services of a distributed operating system such as LOCUS. The approach, however, is not to offer a familiar user interface, as does LOCUS, but rather to attempt to *layer* the services on top of an existing operating system. It is with this approach, that implementations will be simplified and the debugging period will be greatly shortened.

Users who neither need nor desire the fault tolerance services should be able to operate as usual, with no ill effects (including no performance degradation when the services are not accessed). The aim is to make these services as conveniently accessible as possible.

3.5 Layered Distributed System FT Characteristics

This research involves the design of protocols to layer fault tolerance features upon general purpose loosely coupled networks, and to implement these protocols. This means that a selection of services must be chosen which, while not necessarily complete, can be implemented and tested to demonstrate the validity of the approach. The following is a list of fault tolerant features which will provide such an environment:

1. Disk mirroring. While minimization of hardware replication is desirable, the rate of hard errors on disk drives far exceeds that of processors and other components, especially in non-optimal

physical environments. The option to mirror a disk at one or more nodes allows for backup and immediate access to *critical data*. Data is considered *critical* to an application if the data cannot be reproduced in a timely fashion and the ability of the application to perform as required is substantially reduced in the absence of the data.

2. A general loosely coupled network. Because this research aims to layer fault tolerance services upon an existing software environment with no additional hardware, the system must already exhibit the ability to continue operation in the face of hardware failures. The isolation between independently functional nodes provided naturally by a loosely coupled network fills this requirement.
3. A high level of node interconnectivity. This allows a high degree of continued connectivity in the face of node and link failures. In addition, the lower network distance results in smaller message hop counts.
4. Node independent message routing. This is required if the system is to continue operation in the face of node failure, such that the location of tasks cannot be assumed to remain static.
5. Task relocation. While it is not feasible to relocate failing tasks completely transparent to the restarted task, the objective is to provide services to make restart as convenient as possible.

3.6 Fault Tolerant Monitor (FTM)

The general approach is to provide a Fault Tolerant Monitor to run at each node (Figure 12). Each FTM will be transparent to any applications which do not specifically access its services. These FTMs will

communicate with each other across existing communication links between the nodes. This horizontal distribution allows an application task at a node to request a service from the local FTM, which in turn will arrange for the service to be provided through the FTM on the node at which the service is available. All server and client tasks need only register with the FTM running on the local node.

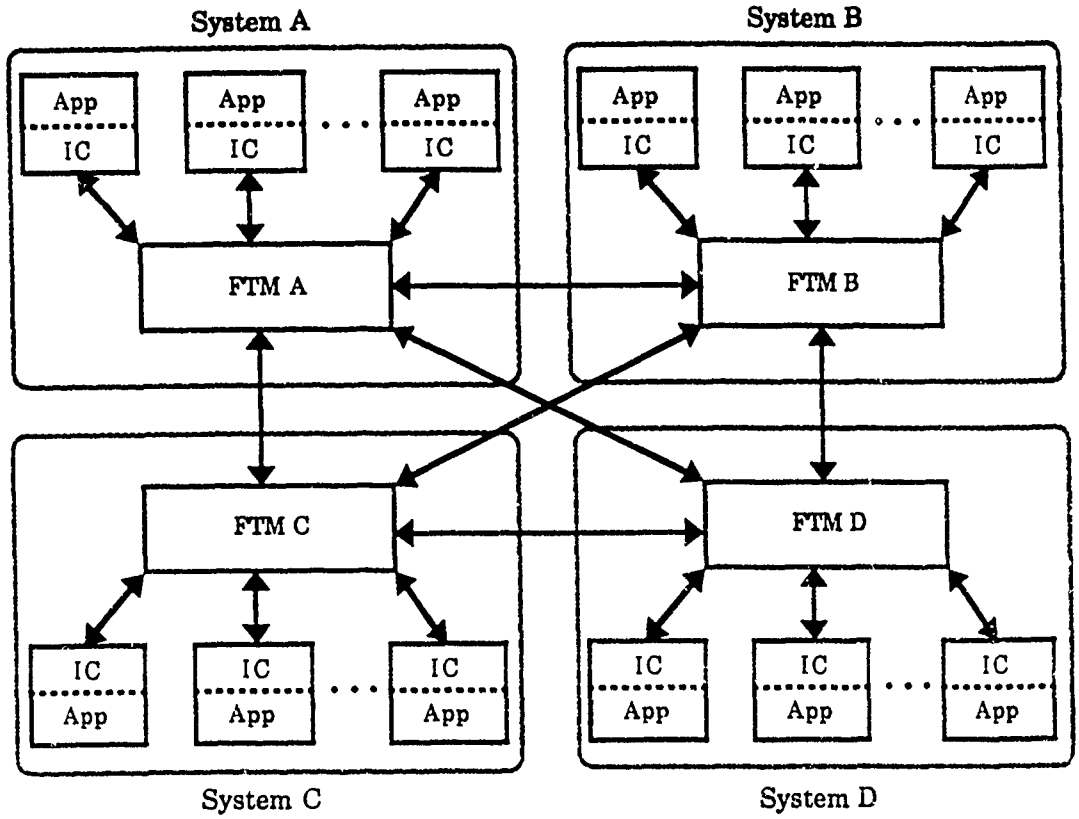

Figure 12. FTM System Overview

3.7 Intercept Code (IC)

While the details of communication between the FTMs are of little concern to the applications programmer, the application must communicate with its local FTM. To simplify this communication and to isolate the

application programmer from the underlying application-to-FTM protocols, an intercept library is provided. This library need only be linked into the application along with the standard libraries. The applications then simply make calls to routines within this intercept library which will handle the details of message passing back and forth with the local FTM.

The term *intercept* code (and *intercept* library) are currently inaccurate. Originally, the intent was to rename standard library calls so as to invoke FTM services completely transparent to the application (by *intercepting* standard library calls and forwarding the request to the FTM system.) This concept evolved to the current "visible but convenient" approach, but the term *intercept* seems to have stuck.

3.8 Overview of Services Offered

Applications dynamically invoke FTM system services through calls to various user interface routines in the Intercept Code. These routines are made available to the application by the system linker, which includes the Intercept Library in the application executable module (Figure 13).

It is the job of the Intercept Code to convert these calls from the application to the user interface routines into IC-to-FTM protocol messages. Appendix A describes all FTM system protocols in detail. The bulk of the fault tolerance services are provided by the local and remote FTMs, with the Intercept Code mainly responsible for the interface between the applications and the FTMs.

Appendix B contains a distribution library for the FTM system, consisting of a Readme file followed by Makefile, sample data files, and the complete source for the FTM and Intercept Code. The Readme file includes a complete and detailed user interface description. What follows

here is an introduction to the user interface philosophy along with a description of several FTM services central to application physical location independence.

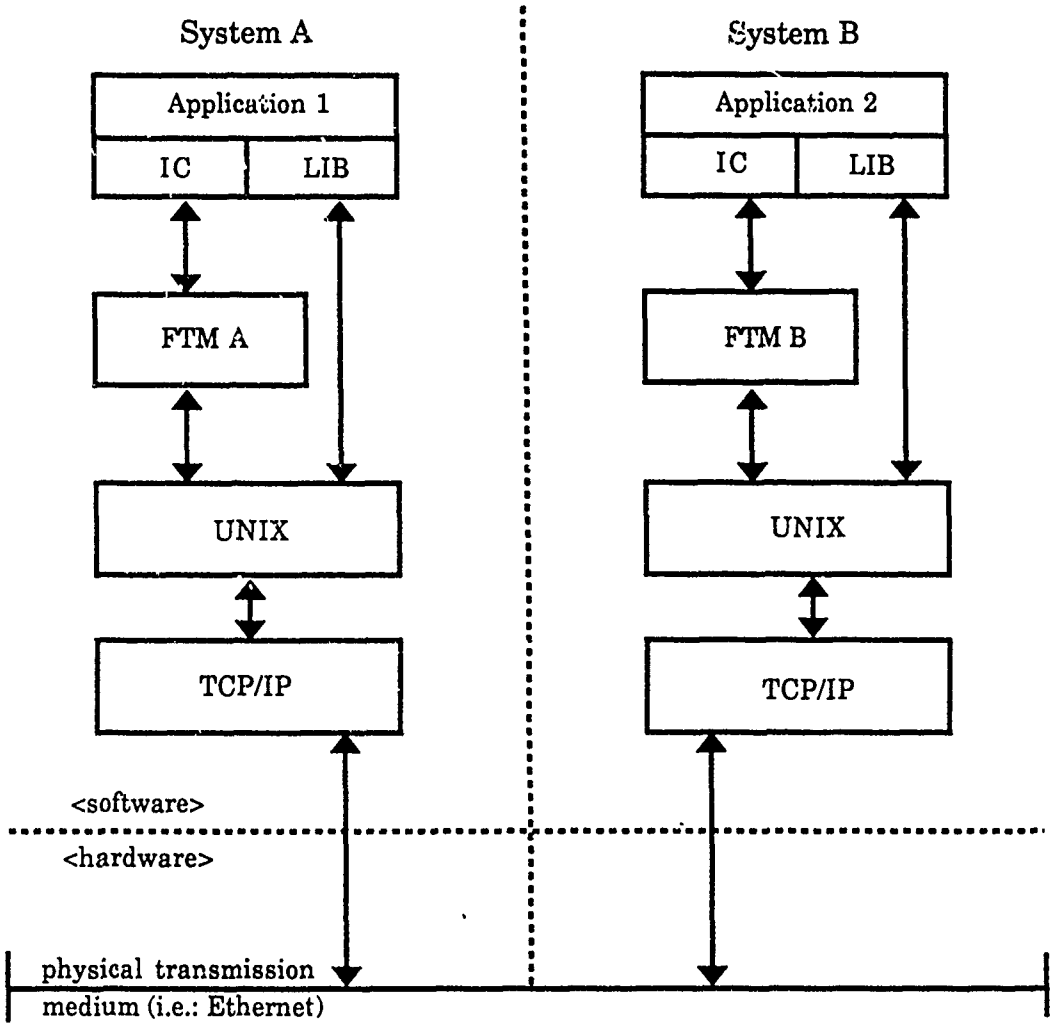

Figure 13. FTM System Communications

3.8.1 Connect to FTM System

The initial step an application must take is to register itself with the local FTM by *advertising* a name. The local FTM will assure that this name is unique throughout the distributed FTM system. This unique

name serves as the identification tag for this application. Other applications which must communicate with or monitor this application may do so through its unique advertised name. In this way, the physical location of an application is immaterial to other applications which access it through the FTM system.

3.8.2 Message Delivery

Applications which have advertised themselves to the FTM system may exchange messages with each other. Since each advertised name is unique within the FTM system, and since the FTMs keep track of the physical location of each task which has advertised, a task need only specify the advertised task name to which messages are to be forwarded. The FTM system will deliver messages to the physical node upon which the target application currently resides.

3.8.3 Task Relocation

An application which is to be restarted at an alternate node, because of failure at the primary node, can be entered into the FTM backup database. In the case of primary failure, the FTM system will restart the application at the backup node. It is not realistic to make this restart completely transparent to the restarted task, so the restarted task may be required to execute some reinitialization code. The new physical location will be broadcast through the FTM system, so that all other communicating tasks will forward messages to the correct current node in a transparent manner.

3.8.4 Critical Files

In the case of a node failure, data present at that node becomes

unavailable. So, even though a task may be relocated it may not be able to operate properly at its new node due to lack of current data. The FTM system therefore recognizes requests from applications to define files as *critical*, in which case the files are copied to the designated backup node for that application.

3.8.5 Status

While the philosophy of the FTM system is to insulate applications from the need to be aware of the current topology of the FTM system or the physical location of other applications, there may be instances where this information is useful for application level monitoring or performance factors. Applications may therefore request various current status information from the FTM system.

3.8.6 Disconnect from FTM System

The final step an application must take is to disconnect itself from the FTM system by *closing* its connection to the local FTM, which in turn will propagate this disconnection throughout the FTM system. If an application terminates after connecting to the FTM system and before disconnecting, then the FTM system will consider the termination to be abnormal. In this case, the FTM system will check its restart database and, if possible, will attempt to restart the application on its designated backup system.

3.9 Introduction to User Interface

Appendix B includes the programmers manual for the complete user interface. Presented here is a basic description of the minimal subset of IC procedure calls required for an application to avail itself of the

FTM services. This minimal subset consists of four routines:

adv (MyName)

from (HisName, MsgBuffer)

to (HisName, MsgBuffer)

ftmclose ()

MyName is the system-wide unique name by which the calling application will be identified within the FTM system. HisName is the system-wide unique name by which a communicating application is identified within the FTM system. MsgBuffer contains a message (or the space for a message). The message may be free-form. Header fields will be prefixed and stripped by various components of the FTM system en route so that the final delivered message is a duplicate of the original message.

3.9.1 Globally Advertise a Name

A typical application program would interact with the FTM system by first calling adv (MyName). The IC linked into that application will attempt to connect to an FTM running on the local node. Assuming successful FTM connection (otherwise the IC returns immediately with an error code), the IC will next format a message (using IC – FTM protocol) and send it to the local FTM. The local FTM will check MyName against its database of current advertised names. Assuming that MyName is not a duplicate name, the FTM will then attempt to globally register MyName by broadcasting the request to all other FTMs (using FTM – FTM protocol). A 2-phase commit scheme (explained shortly) is employed to handle the case where distinct FTMs are simultaneously attempting to register the same name. The success or failure of the global registration attempt is then reported back to the IC by the local FTM. The IC finally returns to the calling application with a code to indicate success or failure.

The 2-phase commit protocol assures that no FTM accepts MyName as unique until all FTMs agree. Consider the following 1-phase commit scenario: We have 4 FTMs. Say FTM_1 and FTM_2 simultaneously attempt to globally advertise MyName. FTM_1 sends Myname to FTM_3 and FTM_2 sends Myname to FTM_4. FTM_3 and FTM_4 each reply that the name has been registered. As the broadcasts continue, FTM_1 and FTM_2 will eventually receive return messages indicating duplicate name, and de-register the name from FTM_3 and FTM_4. There is a period of time during which FTM_3 and FTM_4 each believe that MyName is a unique global name. Actions taken during this period may be difficult or impossible to roll back.

The 2-phase commit solution is to make the acceptance of MyName conditional. In the above example, FTM_3 and FTM_4 would conditionally accept MyName (since it is unique to each of them). But, the conditional acceptance does not become permanent until the original sender so instructs. The original sender, in turn, will not attempt to broadcast instructions to make MyName permanent until it has received conditional acceptance from ALL other FTMs. In the above example, FTM_1 and FTM_2 will eventually receive two duplicate name messages (one from each other and one from either FTM_3 or FTM_4). As each of FTM_1 and FTM_2 receives its duplicate name message, it instructs the other FTMs to delete MyName rather than to make it permanent. So, at no time would any of the FTMs consider MyName to be globally unique.

3.9.2 Receive a Message

In order to receive a message, the application program need only execute a call to the IC routine: from (HisName, MsgBuffer), in order to receive the oldest pending message from some application program which

has globally advertised the name: HisName. Options to the call allow the caller to receive the oldest message from any sender. The IC routine does not wait for a message to arrive. Rather, it returns immediately with a status code to indicate the length of the message received. A status code of -1 indicates that a further code should be examined by the calling program to distinguish between no message or some error condition.

3.9.3 Send a Message

A message may be sent to another application through the IC call: to (HisName, MsgBuffer). A successful return code indicates that the message has been received by the IC of the receiving application, although this does not guarantee that the receiving application will ever get around to processing it.

For both message operations (send and receive) neither application needs to know the physical node on which the other resides. The FTMs will manage the physical locations and redirect message traffic as required.

3.9.4 Disconnect from the FTM System

A call to the IC routine: ftmclose () results in the caller's globally advertised name being deleted from the system. This call is optional, except for one case: if the calling application has been included in the FTM system's restartable/relocatable database, and if the application advertises a name, and if the application then ends (normally or abnormally) without a call to ftmclose (), then the FTM system will attempt a restart of the application.

3.10 Protocols

Despite similarities in message format and contents, there are actually three distinct protocols involved in the FTM system: an IC-to-FTM (IF) protocol, an FTM-to-FTM (FF) protocal, and an FTM-to-IC (FI) protocol. A message in any of the three protocol classes follows the following format (defined as a structure in the C language):

```
struct msgbuf {
    char  cmd[7];          /* message type */
    char  from;            /* message from FTM or IC ? */
    char  status[6];       /* action taken on msg so far */
    char  name_to[21];     /* target of message */
    char  name_from[21];   /* originator of message */
    char  seqnum[12];      /* message sequence number */
    char  errno[12];       /* error code for exceptions */
    char  mlen[12];        /* length of the next field */
    char  msg[2049];       /* data portion of the msg */
}
```

A general discussion of the protocol message contents is offered here; a detailed discussion of each protocol message type appears in Appendix A.

cmd – The protocol message type. See Table I for a brief description of these types. They will subsequently be described in detail.

from – The message source (F for FTM and I for Intercept Code). Since messages must sometimes be stored for later action, and since the same message type can require different actions depending on whether received from an application's Intercept Code or from another FTM, the message source must be maintained until the message has been fully acted on.

status – The current status of a message. This is always -1 (immediate action) for a message as received. However, as mentioned above, the message may require temporary storage while a higher priority action

is taken, or until an action taken because of the message produces a result. This field defines the next action required in support of the message.

name_to – The advertised name of the application whose Intercept Code is the final target of this protocol message.

name_from – The advertised name of the application whose Intercept Code is the originator of this protocol message.

seqnum – The sequence number associated with this particular protocol message. Used by the FTM system internally to assure that protocol messages are not lost in transit and that protocol message echo-backs are compared to the correct original message.

errno – Error return code. If a protocol message is in response to a past protocol message which could not be handled correctly, this code identifies the error.

mlen – The length of the following (variable length) field. While the protocol-defined fields avoid the use of non-ascii characters, the following field may contain arbitrary user data so that a predefined terminator character does not exist. The use of this message length field avoids the requirement that a duplication scheme be implemented to define the end of the user data field.

msg – Variable length data. This can contain certain FTM system data, or, for certain protocol message types it will contain user data.

The message types (CMD) and protocol classes (PC – IF for IC-to-FTM, FF for FTM-to-FTM, and FI for FTM-to-IC), followed by a brief description of each type, is given in Table I. Appendix A contains a detailed discussion for each type.

Table I. IC-to-FTM, FTM-to-FTM, and FTM-to-IC Protocols

FTM SYSTEM PROTOCOLS		
CMD	PC	DESCRIPTION
ADV	IF	IC tells local FTM to advertise a name
ADV	FF	FTM attempts to globally advertise
ADVOK	FF	Phase-1 positive acknowledgement
ADVOK	FI	FTM informs IC of successful advertise
ADVNO	FF	Phase-1 negative acknowledgement
ADVNO	FI	FTM informs IC of unsuccessful advertise
ADVOKY	FF	Phase-2 positive acknowledgement
ADVOKN	FF	Phase-2 negative acknowledgement
CLOSE	IF	Request to De-Advertise a global name
CLOSOK	FI	De-Advertise completed
KILLN	FF	FTM broadcasts application termination
STATUS	IF	IC requests status info from local FTM
STATOK	FI	FTM replies with status info
TO	IF	User msg is sent: IC –> local FTM
TO	FF	User msg is sent: local FTM -> remote FTM
TO	FI	User msg is sent: remote FTM -> IC
TOOK	IF	Usr msg delivered to remote IC
TOOK	FF	Next hop on return path
TOOK	FI	Final hop on return path
TONO	IF	User msg not delivered to remote IC
TONO	FF	Next hop on return path
TONO	FI	Final hop on return path
WHERE	IF	IC requests current node for some name
WHEROK	FI	FTM replies with node name

CHAPTER IV

FTM IMPLEMENTATION

4.1 Introduction

In order to test the design of the FTM system protocols, it is necessary to implement a distributed system employing these protocols. In this chapter such an implementation is described.

4.2 Implementation Overview

The Fault Tolerant Monitor software system (see Figure 12, Chapter III) contains two distinct parts, connected only through message communications. These are: the Fault Tolerant Monitor (FTM) and the Intercept Code (IC). Together, they implement the required functionality.

First, there is the FTM itself. One FTM will be present on each node. These FTMs communicate with each other via messages, passing information necessary for each to maintain a current and consistent view of the distributed systems. The FTMs also monitor each other. An aberrant or failed FTM that is detected by some other FTM will be disconnected from the group and any eligible applications currently under the control of the failing FTM will be relocated to a node with an active FTM.

The IC routines can be present in multiple applications on a single node. Each IC is linked into an application, along with the standard library routines. Each application makes procedure calls to the IC user interface. The IC, in turn, sends and receives messages to/from the local FTM on its node. This insulates the application programmer from the implementation details, since the programmer needs to understand neither the IC-to-FTM protocols, the FTM-to-FTM protocols, nor the FTM-to-

IC protocols.

4.3 Software Choices

While the protocols of this research are language and software environment independent, it was necessary to choose a specific software environment in which to implement the prototype. Since hardware independence is also a goal, UNIX is an obvious choice for an operating system. The aim is to offer some of the functionality of a fault tolerant UNIX-like operating system, such as LOCUS, but to implement this mostly by layering routines on top of standard UNIX. Modifications to UNIX itself are limited to disk drivers for the support of disk mirroring.

The chosen implementation language is C. While C is promoted as a language which is independent of the operating system, the FTM system implementation is dependent on operating system specific calls for such services as memory management, inter-process and inter-processor communication, sub-process creation, and current environment characteristics. This dependence precludes ease of translation for the FTM system from UNIX to any non-compatible operating system, despite the choice of C as the implementation language. Rather, C was chosen because of its power to manipulate arbitrary data structures in a straightforward manner. A system such as the FTM system is highly dependent on the efficient implementation of a variety of data structures for internal use, for interfacing with applications which may be written in a variety of languages, and for calls to UNIX.

4.4 Hardware Choices

The decision to implement the prototype under UNIX has been fortuitous. The initial hardware environment was a network of VAXes

running ULTRIX, Digital Equipment Corporation's version of UNIX. An abbreviated version of the initial prototype was demonstrated on this system. Then, for convenience of access, this was ported to a network of SUN 3/50s and 3/160s (based on Motorola 600X0 series microprocessors). Prototype development continued and a subsequent version of the initial prototype was demonstrated on these SUNs.

Next, the prototype was ported to hardware owned by a government agency interested in the concept of layered fault tolerance. This hardware is a network of SUN 386i workstations (based on the INTEL 80386 chip). Again, prototype development continued. The final version of the initial prototype was successfully demonstrated on this SUN 386i network.

After the successful demonstration, the government agency decided to continue research into fault tolerance layered over UNIX. The SUN 386i network was replaced by a SUN 4 network. The SUN 4 family is based on SUN Computer Corporation's SPARC microprocessor, which is SUN's implementation of reduced instruction set (RISC) computer architecture. The FTM system has been hosted on this SUN 4 network and is presently undergoing further development. This SUN 4 system is used as the platform for testing and performance data, as presented in Chapter V.

4.4.1 Disk Mirroring

Disk mirroring has been identified as one area in which optional hardware replication enhances system reliability and its effect overshadows the cost of hardware duplication.

The aim is to allow the FTM system manager the option to maintain duplicate data at a specific node if deemed advisable based on factors such as the delay involved in relocating processes should the data become

unavailable at that node. The low reliability of disk drives (as compared to the central system reliability) points to the disk drive as a hardware weak link. A disk error could mean that data must be recreated or recovered from a remote disk. Or, a hard disk error could make application relocation necessary even though the only error condition at a node is at the disk drive.

One option is to allow the FTM system to manage all application data. The FTM system could then maintain a mirrored file system, duplicating all such data on the same disk, on another local disk, or at a remote disk. The choice would be based on disk availability and allowable recovery time.

A drawback to this approach is the lack of efficiency. Since the FTM is layered on top of UNIX, this forces an extra layer between the application and the operating system for I/O. Also, this approach deals only with the application data. If the error occurs on other data, such as operating system data, FTM data, or even paged memory transfers, then the FTM data duplication is insufficient.

The approach toward data duplication at a node is to layer the support *beneath* UNIX, rather than on top of UNIX. The disk driver is modified so that a write to a certain disk drive results in an additional identical write to a *shadow* disk drive, which is maintained as an exact duplicate of the primary drive.

This disk driver modification is independent of the rest of the FTM system and may be installed at a node with or without the intention to run an FTM at that node. The only disadvantage is the cost of the shadow disk drive. The advantage is that as long as the data required for

continuous system operation and important processes is assigned to the disk(s) being shadowed it is duplicated with negligible overhead.

The read routines in the modified disk driver normally read from the primary disk. An error causes an automatic retry from the identical block on the shadow disk. The primary disk is tagged as being down, with an information message logged, and future I/O is from/to the shadow disk only.

Since disk mirroring is layered beneath the UNIX operating system it does violate the "easily transportable" concept. Disk mirroring was implemented and tested along with the initial FTM partial implementation on a network of VAXes. Thus, disk mirroring was installed as a modification to the ULTRIX disk driver for RA81 disk devices on a VAX 11/750 computer. ULTRIX is Digital Equipment Corporation's VAX implementation of Berkely 4.2 release 3.5 UNIX.

Disk mirroring has not been included in subsequent FTM implementations because a disk drive could not be devoted to a shadow role. Since this feature is totally independent of the remainder of the FTM system, it could be installed at any time in the future should the requirement for duplicated data at a node arise. Disk drivers are hardware dependent (they are dependent both on CPU hardware and disk controller hardware), and thus the alterations made to the VAX 11/750 disk driver for RA81 disk devices would *not* be easily transportable to other architectures.

4.5 FTM Library

Appendix B contains a complete listing of all text files (including C source files) which comprise the FTM system. This set of files represents

the *distribution library*. From this distribution library it should be straightforward to generate an FTM system to run on any "reasonable" loosely coupled network of UNIX systems. Note that the system generation instructions in the distribution library do assume that the individual network nodes are binary compatible. If this is not the case, then some additional effort must be taken to generate a correct FTM system. This subject is discussed in Chapter VI.

4.5.1 File: *Readme*

This file is an overview of the distribution library, with instructions on how to generate an operational FTM system from the distribution library, and a complete guide to the Intercept Code user interface.

4.5.2 File: *Makefile*

When used as input to the standard UNIX "make" utility program, this file results in the generation of the FTM program, the Intercept Code object library, and several FTM system utility programs.

Another file which can be used as input to the "make" utility is *Lint*. In this case, the "make" utility performs a sequence of "lint" invocations. Program "lint" is a standard UNIX utility which checks for inconsistencies in "C" source code files which may not be detected by the "C" compiler. Program "lint" produces a large quantity of output, most of which is 'possible' inconsistencies which turn out to reflect the intention of the programmer. But, there are times in which "lint" will pick out inconsistencies which are not the intention of the programmer and which would be very difficult to isolate without the "lint" output.

4.5.3 File: *header.h*

This is the only user header file present in the distribution library. All structures and constants common to multiple FTM system modules are defined in this file; thus the coordination of interdependence between modules is simplified. The use of a single large header file can be inefficient, in that a single change in the file causes a recompilation of many individual modules rather than just the few which reference the single changed item.

The single header file also simplifies initial FTM system generation from the distribution library. After setting the defined constants in the header file, a UNIX "make" (described in section 4.5.2) generates the entire FTM system.

4.5.3.1 Global Structure Definitions

The FTM system uses three main structure types. These are defined in the *header.h* file, which is reproduced in Appendix B. A general description is presented here.

The first main structure is called *ftmtable*. Each FTM maintains an *ftmtable* entry for each FTM in the system, including itself. An *ftmtable* entry maintains information about a single FTM, including node name, logical connection for message traffic, and status (up or down). Since each FTM at startup will ascertain the number of other FTMs, the storage for all *ftmtable* entries can be allocated in one block. This allows these *ftmtable* entries to be contiguously stored, simplifying indexing.

The second main structure is called *namtable*. Each FTM maintains a *namtable* entry for each advertised name in the system. A *namtable* entry contains such information as the advertised name, current

connection status, communication path from the FTM to the advertising application (direct or through a remote FTM), and the backup FTM for the application. Since the number of connected applications will vary over time, a linked list is used to organize the namtable entries.

The third main structure is called *msgbuf*. Each entry is an FTM system message (described in section 3.10 and Appendix A) augmented by two pointer fields so that the messages may be maintained in a linked list. Each FTM maintains a list of messages which are requests for service by that FTM. Messages which cannot be serviced immediately are held pending some enabling system event. Each application Intercept Code also maintains a linked list of messages. In this case, the IC is buffering the messages so that the application can receive them at some later time.

4.5.4 File: *ftms.lis*

This is a list of the loosely connected nodes on which the next started FTM system should run. It is simply a sample file, with comments in the heading which explain how to enter node names for the target set of nodes.

4.5.5 File: *bkup.lis*

This is another sample list. In this case, the list is of complete path names which the FTM system may use to restart applications which fail while connected to the FTM system. Again, while this is only a sample file it contains comments in the heading which explain how to format file entries.

4.5.6 FTM System Source Files

This is a series of C source files, printed in alphabetic order, which

comprise the FTM system. This includes the FTM itself, the Intercept Code routines, and utility programs and routines. Refer to the Makefile (described in section 4.5.2) for actual compilation, linkage, and object library creation.

4.5.7 Test Program Source Files

Finally, the C source files with names beginning with the letter 'z' are applications used to test the features and time the response of the FTM system. Data from these test programs is the basis of the results reported in Chapter V.

4.6 Multiple FTMs

In order to allow multiple versions of the FTM system to be tested simultaneously on (possibly) overlapping collections of nodes, each FTM system is hard-coded to a specific path for its initialization data and a specific logical port for its FTM to FTM communications. Application programmers need only assure that the IC linked into their program matches the running FTM system on which they expect to be operating.

In keeping with the aim of simplified system generation, these paths are defined in the file: *header.h* (see section 4.5.3). Several copies of modified distribution libraries could be maintained and as long as each has a unique path defined in its *header.h* file there will be no confusion between simultaneously running FTM systems or applications connected to any of the FTM systems.

4.7 FTM System Startup

In order to initiate an FTM system, it is necessary to access the directory structure for the version desired, update one file in that

directory (*ftms.lis* – a list of the nodes to be included), and execute the stand-alone program: *ftmstart*. This program (*ftmstart*) will invoke a UNIX daemon to initiate an FTM (with program name: *ftm*) on each listed system. The FTMs then interconnect and become ready to accept advertised names from applications on the nodes. Before starting any applications it may be desirable to check the file: *ftmbkup.lis*, which contains complete path names for applications which the FTM system will attempt to restart/relocate in the case of an application (or node) failure.

4.8 FTM System Steady State Operation

Each operational FTM system, once started, will continue to operate until terminated. During periods when no application is interacting with the local FTM, the CPU overhead is very small (<< 1%) and applications can join the system (by advertising a name) at any time. The *ftmbkup.lis* file, which contains complete path names for restartable/relocatable applications, can be modified by users or applications at any time, assuming update conflicts are avoided.

As an FTM becomes unavailable, either because it is terminated, it fails, or its node fails, the other FTMs exclude it from the system and attempt to relocate applications from the failed FTM to an operational FTM. The FTM system is down only when the final FTM goes down. A link failure could cause an FTM system partition, in which case each partition would attempt to restart/relocate applications on its own. Consensus mechanisms to avoid this anomaly will be included in future FTM system prototypes.

4.9 FTM Logic Flow

The mainline FTM program appears in the file: *ftm.c* (Appendix B).

One instance of this program runs on each of the nodes in the distributed FTM system. Each of these FTM programs basically loops forever responding to requests from the others, and to requests from local connected applications. Figure 14 outlines these FTM actions.

4.9.1 Connect to other FTMs

Upon startup, each FTM immediately attempts to connect to all other FTMs. Subroutine: *conall* (file: *conall.c*) is provided for this purpose. Once *conall* has determined the number of FTMs to be included in the current configuration (from file: *ftms.lis*) it can allocate the storage to hold the consolidated collection of *ftmtable* structures (see section 4.5.3.1). Next, the *ftmtable* structures are filled as connections to other FTMs are completed. Since UNIX socket connections are asymmetric, each pair of FTMs must coordinate which will originate the connection request. The algorithm used assumes that the file: *ftms.lis* available to each of the FTMs is identical in the order of nodes included. So, if the FTMs are not sharing the file through a network file service (NFS) then the copies of the file that are used must not deviate in order of contents.

4.9.2 Accept IC Connections

Assuming that subroutine: *conall* has successfully connected to all other FTMs within the timeout period, the FTM now enters its main processing loop. The initial step within this loop is to check for pending connection requests from applications on the local system. In the asynchronous UNIX socket environment, the FTM acts as the server while applications act as clients, requesting connection to the FTM as needed. Unlike the FTM-to-FTM connections, where the connection topology is preordained by the contents of the file: *ftms.lis*, each FTM does not know nor

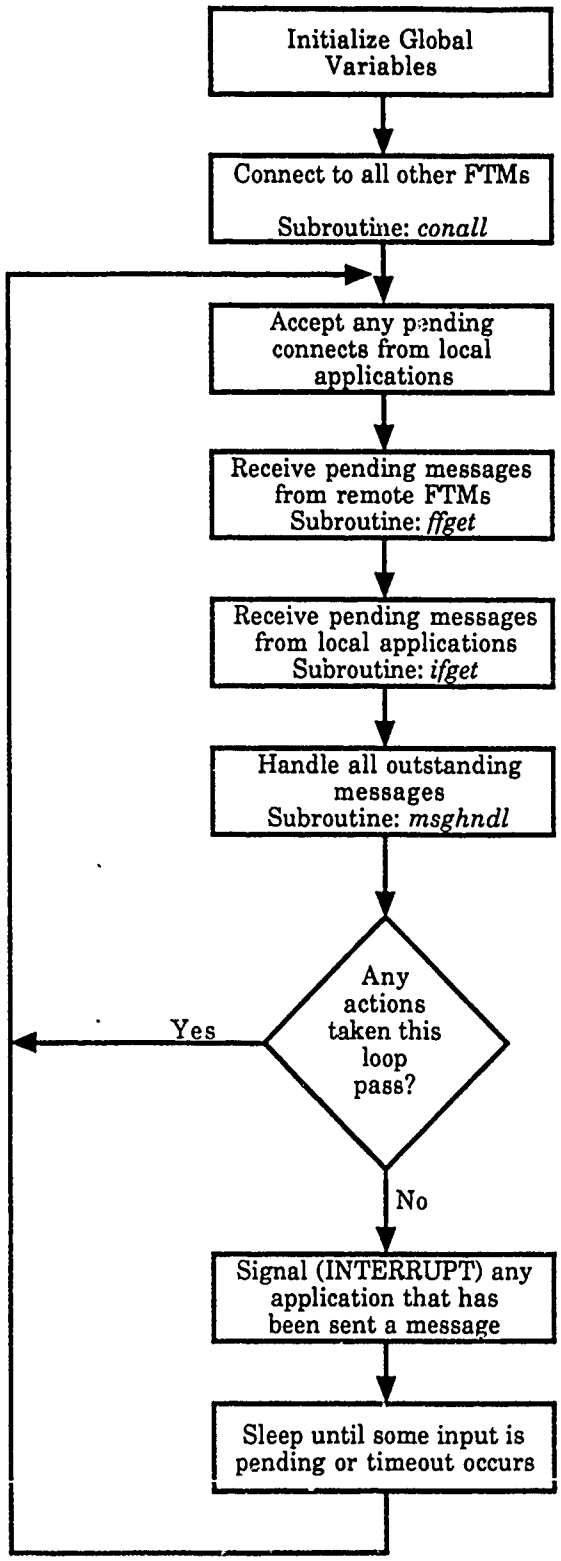

Figure 14. FTM Main Loop

care which applications will want to connect to it. So, in this case it must be the application which initiates the connection request.

As each pending connection from a local application is accepted, the FTM allocates space for a *namtable* structure (see section 4.3.5.1) which will hold information about the application. This allocated structure is linked onto a list (using subroutine: *namtolist*) which contains one *namtable* structure for each connected application (local and remote) throughout the FTM system. Since this information must be available throughout the FTM system (for message routing and application restart) other FTMs must eventually create similar *namtable* structures for their own internal use.

The connection request contains minimal information, which does *not* include the advertised name, so the propagation of the *namtable* information throughout the distributed FTM system must be delayed until it becomes available to the local FTM. The first message over the just opened communication socket will be a request from the application's Intercept Code to advertise a name. At that time, the skeletal *namtable* structure can be completely filled in and the information propagated to other FTMs.

4.9.3 Read Messages from other FTMs

The next step in the main FTM loop is to read any pending messages from other connected FTMs. This is accomplished by a call to Subroutine: *ffget* (file: *ffget.c*). While a minor amount of preprocessing is performed for certain protocol message types, the main function of *ffget* is to queue any messages read. First, a structure *msgbuf* is allocated (see section 4.5.3.1). The preprocessed message is augmented into this allocated

msgbuf structure, which is then added to the linked list of messages.

This step also serves as a check on the current connectivity of other FTMs. If a remote FTM is found to no longer be accessible, then the *ftmtable* structure for that unavailable FTM is modified to indicate this status. In addition, subroutine: *ftmisdown* (file: *ftmisdown.c*) is called to handle the failure.

4.9.4 Read Messages from Connected ICs

After reading any pending messages from other FTMs, the local FTM next reads any pending messages from local connected applications. Subroutine: *ifget* (file: *ifget.c*) performs this function. Again, there is a certain amount of preprocessing involved, but the main purpose of *ifget* is to queue the messages read onto the linked message list.

The continued connectivity of each connected application is checked during this step. Recall that if an application becomes unconnected after advertising a name but before issuing a request to close, then that application is considered by the FTM system to have terminated abnormally. In this case, the *ifget* subroutine will construct and queue a KILLN FTM-to-FTM class message, which will eventually be propagated to the other FTMs informing them of the abnormal termination. If appropriate, the FTM at the backup system will then attempt a restart.

4.9.5 Handle Pending Messages

The final processing step within the main FTM loop is to "handle" the messages which have been queued onto the message list. Subroutine: *msghndl* (file: *msghndl.c*) is called at this point. The *msghndl* subroutine scans through the message list, searching for messages which require

immediate action. The status field of each message defines the next action (if any) to be taken for the processing of that message (see section 3.10). As *msghndl* finds a message which requires processing, it calls the appropriate message "handler" subroutine. These are all named *hndlxxxx*, where *xxxx* is a reasonably mneumonic string, and can be found in Appendix B (in file: *hndlxxxx.c*).

Table II provides a list of the *hndlxxxx* subroutines, along with the associated types of message. The message types are listed as tuples: <xxxx,y>, where xxxx is the CMD message field and y is F for an FTM-to-FTM class message or I for an IC-to-FTM class message. Both elements are required to uniquely identify a message type (see section 3.10).

Table II. FTM Message Handling Routines

FTM MESSAGE HANDLING ROUTINES	
ROUTINE	MESSAGE TYPES HANDLED
hndladvf	<ADV,F>
hndladvi	<ADV,I>
hndladvok	<ADVOK,F>, <ADVNO,F>
hndladvoky	<ADVOKY,F>, <ADVOKN,F>
hndlclose	<CLOSE, I>
hndlkilln	<KILLN,F>, <KILLN,I>
hndlstatus	<STATUS,F>
hndlto	<TO,F>, <TO,I>
hndltook	<TOOK,F>, <TOOK,I>, <TONO,F>, <TONO,I>
hndlwhere	<WHERE,I>

It is significant that *msghndl* scan the message list in order. Certain of the *hndlxxxx* subroutines use this order to queue messages when future processing must be accomplished in that sequence.

The actions taken by the *hndlxxxx* subroutines will be described briefly. The C code in Appendix B may be referred to for further details.

The *hndladvf* subroutine acts on ADV messages from other FTMs. The other FTM has received a request from one of its local applications to advertise a name, so that FTM is attempting to find out if the name is unique throughout the distributed FTM system. The *hndladvf* checks to see if the name has been used locally. This could happen, for example, if a local application was in the process of attempting to advertise the same name. In this case, the *hndladvf* subroutine will construct an ADVNO message and return it to the remote FTM.

Assuming the name does not conflict, the *hndladvf* subroutine will create a *namtable* structure for the remote application, and link this structure into the list of system-wide names. However, the status in this entry will indicate that the entry is tentative, since only phase 1 of the 2-phase commit protocol has been completed. Phase 1 (section 3.9.1) only assures that the name is unique to the remote and local FTMs. Phase 2 is required to assure that the name is unique throughout the FTM system.

The *hndladvi* subroutine acts on ADV messages from local applications. When the application calls the *adv* Intercept Code user interface routine (as will be described in the next section) two actions occur. First, the IC attemps to connect to its local FTM, as described in section 4.9.2. Next, the IC sends to its local FTM one of the <ADV,I> messages. The *hndladvi* subroutine checks to see if the advertised name is unique to it. If not, *hndladvi* constructs an ADVNO message and sends it to the requesting IC.

If the advertised name is unique to the local FTM, then the local FTM enters phase 1 of the 2-phase commit protocol. It constructs an FTM-to-FTM ADV message and sends a copy to each of the other FTMs. The <ADV,I> message is then altered so that its status field indicates the

partial processing.

The *hndladvok* subroutine acts on ADVOK and on ADVNO messages from other FTMs. As described above, these messages are received as the result of phase 1 of an attempt by this FTM to advertise a name. If 1 or more ADVNO messages are received for a particular name, then at least 1 remote FTM has rejected the name. So, the *hndladvok* subroutine constructs an ADVNO message and sends it to the requesting IC. The application will as a result be informed that it cannot advertise that particular name at this time. The *hndladvok* subroutine also constructs an ADVOKN message, which it sends to each remote FTM. This is the negative phase 2 message, telling remote FTMs that the advertise has failed and that the preliminary *namtable* structure entries for this name should be discarded.

If an ADVOK message has been received from each of the remote FTMs, then phase 1 has completed successfully. In this case, the *hndladvok* subroutine constructs an ADVOKY message, which is sent to all remote FTMs. The name is now accepted by all FTMs as having been successfully advertised throughout the distributed FTM system.

The *hndladvoky* subroutine handles ADVOKY and ADVOKN messages from other FTMs. These represent the affirmative and negative phase 2 acknowledgements, respectively, as described above. The action taken by *hndladvoky* is to complete (ADVOKY) or delete (ADVOKN) the preliminary *namtable* structure entry for the particular advertised name.

The *hndlclose* subroutine takes care of CLOSE messages. These are requests from the IC of connected applications wishing to disconnect from the FTM system. The actions taken by *hndlclose* are to delete the

appropriate *namtable* structure entry, construct a KILLN message to be broadcast to the other FTMs, and to contruct a CLOSOK message to return to the requesting local application. The KILLN message includes an indicator (in the status field) to report that the application termination should be treated by the FTM system as "normal." That is, no attempt to restart the application should made.

The *hndlkilln* subroutine acts on <KILLN,F> and <KILLN,I> messages. It should be noted that all KILLN messages are actually originated by FTMs, never by Intercept Code. However, if the KILLN is the result of a CLOSE request from some application then the status field is set to 'I'. A status field value of 'F' indicates that the KILLN was originated by an FTM as the result of an abnormal application termination.

In the case of a normal application termination, the local *namtable* structure entry for that application is deleted. Any pending messages destined for that application are deleted from the message list. If the termination was abnormal, then in addition to the above actions a restart/relocation is attempted. This is controlled by information saved from the just deleted *namtable* structure entry, defining the current backup system, and from the file: *bkup.lis* which contains path information for applications which are eligible for restart.

The *hndlstatus* subroutine responds to STATUS request messages. The action taken is to construct a STATOK message, fill in status information from the local FTM database (the *ftmtable* structures for current FTM system information), and send the STAKOK message back to the IC of the requesting application.

The *hndlto* subroutine acts on TO messages, both <TO,F> and <TO,I> message types. In either case, the purpose is to forward the message along to the next hop in the path from the sending application to the intended receiving application. A sending application can route a user message to a receiving application by specifying the advertised name of the receiver (swection 3.9.3). The user message is encapsulated into a <TO,I> message from the application IC to the local FTM. The local FTM then determines whether the receiving application is also local. If so, then the user message can be delivered directly. Otherwise, it must first be forwarded to the appropriate remote FTM, from where it can then be delivered to the receiving application.

When *hndlto* processes a <TO,I> message it constructs a <TO,F> message from it. If the receiving application is local, the <TO,F> message is sent directly to it. If not, the <TO,F> message is instead sent to the FTM local to the receiving application. When *hndlto* processes a <TO,F> message the result is similar, except that since the message has been forwarded from a remote FTM the receiving application will be local. However future, more general, versions of the FTM system will include FTMs which are not fully connected. In this case a message may pass through intermediate FTMs, so a <TO,F> may need to be forwarded to another FTM.

The *hndltook* subroutine handles TOOK and TONO messages from both Intercept Code and remote FTMs. In the case of TOOK messages, the user message imbedded in a TO message has been successfully delivered to the Intercept Code of the receiving application. The IC has queued the message, so the receiving application can read it whenever it so desiresr. The positive acknowledgement (the TOOK message) is then

routed back to the sending IC. At each step, the seqnum field (message sequence number) and user message portion are compared with the saved TO message in order to assure accurately deliverd data.

Similarly, a negative acknowledgement (the TONO message) is routed back to the sending IC. In this case, the sending application is informed that the message was not delivered successfully, with a reason code in the errno field of the TONO message.

The *hndlwhere* subroutine is very similar to the *hndlstatus* subroutine; *hndlwhere* responds to application WHERE requests while *hndlstatus* responds to application STATUS requests. In the case of *hndlwhere*, a WHEROK message is constructed. This message includes data about a specific application somewhere in the FTM system. As with *hndlstatus*, the information is obtained from the FTM system data within the local FTM. The WHEROK message is then sent to the IC of the requesting application.

4.9.6 Loop Control

Because the FTM is interrupt driven, it *sleeps* (i.e., suspends itself) during periods when there are no pending useful actions it can take. The algorithm used is as follows: if during the current pass through the loop any actions have been taken (accepts, FTM messages, IC messages, queued messages handled) then this is considered a high activity period and the loop is immediately re-entered. Some action later in the loop during the handling of messages may have enabled another pending action.

If no actions were taken during the current pass through the loop, then the FTM will sleep until it has a pending message receipt or connection request. But, first it will send a signal (i.e., an interrupt) to each

local connected application to whose Intercept Code the FTM has sent a message since the previous signal was sent. The purpose of this signal is to force the IC to immediately queue these messages and to construct a response back to the FTM. In this way, even though the application may be busy processing and may not try to read the message for some period of time, the FTM will not have to wait for that period of time before it receives an acknowledgement that the IC has received the message. This reduces delays in the FTM system, since the message received acknowledgement (the TOOK message) can make its journey back to the sending application without waiting for the receiving application to actually issue the request to read the message.

After the FTM has been awakened, either through a message receipt, a connection request, or the expiration of a timeout period, it reenters the main loop. If no actions are taken during this pass through the loop then the FTM will immediately begin another sleep period. Constants to define the sleep timeout period and many other FTM and Intercept Code timing factors are defined in file: *header.h* (see section 4.5.3).

4.10 IC Logic Flow

Unlike the FTM, the Intercept Code is not a program. Rather, it is an object library of routines designed to be linked into application programs. The full user interface provided by the IC library will not be described here – refer to Appendix B, file: *Readme*, for full details. What will be described here is the logic flow through the Intercept Code in response to application calls to the user interface entry points.

For most of the user interface IC services, a message must be constructed and sent to the local FTM. For these services, the logic flow is as

follows: First, signal interrupts from the local FTM are disabled. This is because the IC will be modifying the message list, and the interrupt handler routine also modifies this list as it links received messages onto the list. Before control is returned to the calling application, the signal interrupts will be re-enabled.

After disabling signal interrupts, the IC user interface routine next validates the user supplied parameters. Assuming successful validation, an appropriate message is constructed and sent to the local FTM.

All messages sent from the IC to the local FTM require a message in response from the local FTM. In some cases, the local FTM may construct the return message immediately upon processing the message from the IC. In other cases, the local FTM must first communicate with one or more remote FTMs. Regardless, the IC will await a response from the local FTM before returning to the calling application. Recall from section 4.9.5 that a remote IC responds immediately to a TO message (user data message) without waiting for the target application to actually read the message. This is to minimize the delay before the sending application regains control after it issues the TO message.

After the response has been received from the local FTM, indicating successful or unsuccessful completion of the request, control is returned to the calling application along with a code to indicate success, or a reason for failure. A timeout while waiting for a response from the local FTM is treated as an FTM communication failure, which currently aborts the application.

The only current user interface entry point which does not result in a message being sent to the local FTM is a *from* call. This call is made by

an application when it wishes to read a message from another application. An optional message selection criteria may be specified by the application, or it may simply ask for the oldest available message. In either case, the Intercept Code searches its message list to select a message (if one is available) to return to the requesting application. Again from section 4.9.5, the IC queues messages upon arrival. So, the *from* call simply finds a message already on the message list, dequeues it, and passes it back to the calling application. Even if no message is immediately available from the message list, the IC returns to the application; in this case a return code indicates the absence of an appropriate message.

The signal interrupt handler routine in the Intercept Code is called *getmsgs*. This routine simply reads pending messages from the local FTM and adds them to the IC message list. Besides its use as a signal interrupt handler, *getmsgs* is also called directly by various routines within the IC when a new incoming message is expected. There is no conflict with its use as an interrupt handler routine, since signal interrupts have been disabled within the affected IC routines.

4.11 Example of an FTM Session

An example of a very short FTM session between two applications on a 2-FTM system is presented in Figure 15. Actually, many of the actions could be occurring simultaneously, but they have been shown with time separation to emphasize the cause and effect nature of some of the message protocol types.

Figure 15 should be read in time-increasing order from top to bottom. It represents the following actions between applications 1 and 2 (APP_1 and APP_2) and FTMs A and B (FTM_A and FTM_B):

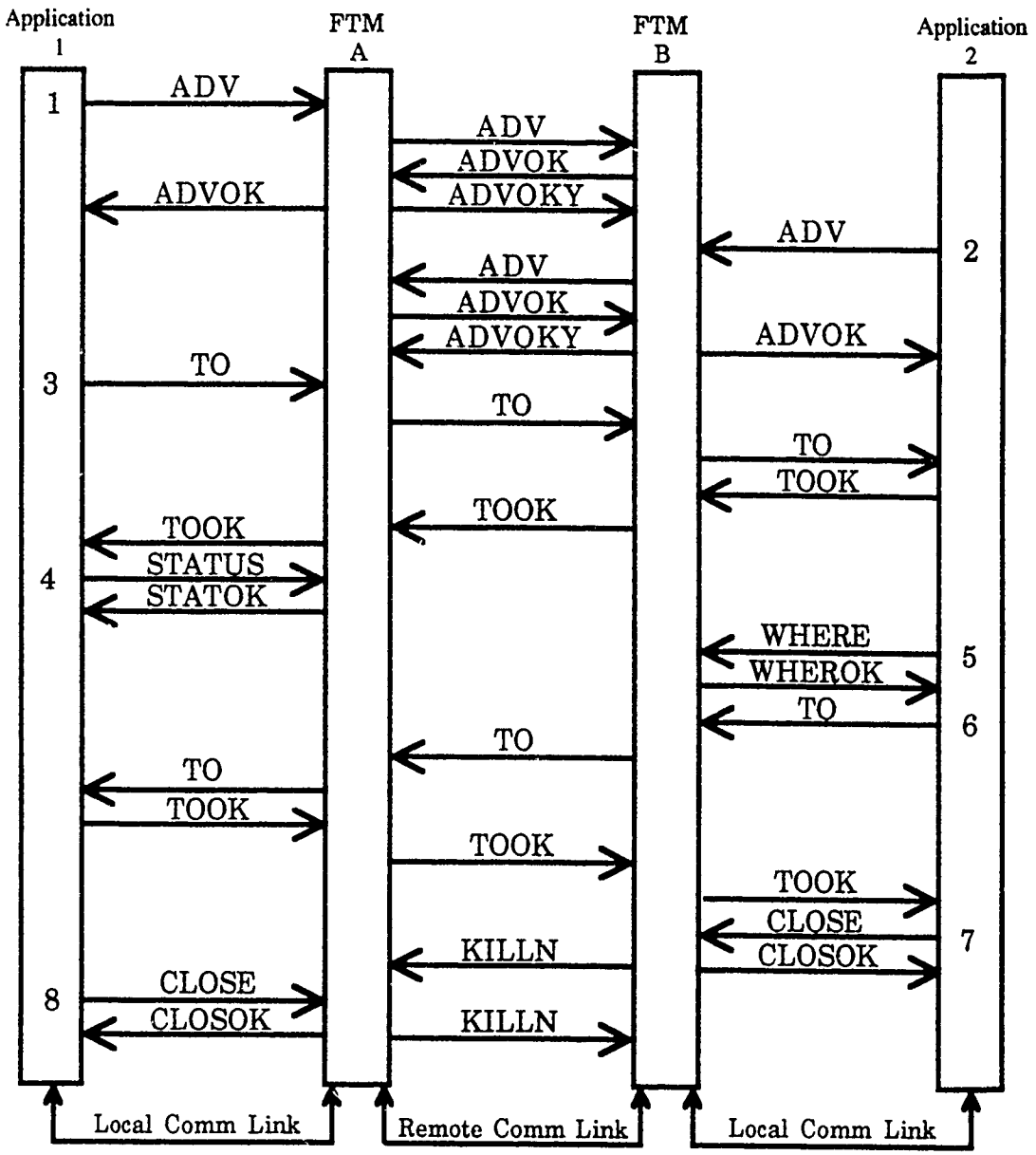

Figure 15. Small Sample FTM Session

1. APP_1 attempts to advertise a name. An ADV message is sent from APP_1 to FTM_A. Next, FTM_A sends an ADV message to each remote FTM, which in our example is just one remote FTM: FTM_B. FTM_B completes phase 1 of the 2-phase commit protocol by returning an ADVOK message to FTM_A. Since FTM_A has

received a positive phase 1 acknowledgement from all remote FTMs (all one of them in our example), the phase 2 positive acknowledgement is broadcast: FTM_A sends an ADVOKY message to all other FTMS, which in our example is FTM_B. At the same time, an ADVOK message is returned from FTM_A to APP_1 which informs APP_1 that the name has been successfully advertised throughout the distributed FTM system.

2. APP_2 attempts to advertise a name. It is assumed that this name is different from the name advertised by APP_1, so that this second advertisement will also be successful. The flow of messages can be seen to parallel that of step 1.
3. After both APP_1 and APP_2 have successfully advertised their names, APP_1 next sends a user message to APP_2. It can be seen that APP_1 sends a TO message to FTM_A, FTM_A responds by sending a TO messsage on to FTM_B, and FTM_B sends a corresponding TO message to the target application: APP_B. Once the Intercept Code within APP_2 has received the TO message, it constructs a TOOK message which it returns to FTM_B. FTM_B sends the TOOK message on to FTM_A, which finally sends the TOOK message to the user message originator: APP_1.
4. APP_1 decides that it would like to know the status of the distributed FTM system. So, APP_1 sends a STATUS message to its local FTM, FTM_A. FTM_A looks up the information in its data base, and returns a STATOK message with the requested information to APP_A.
5. APP_2 decides that it would like to know where APP_1 is located. So, APP_2 sends a WHERE message to its local FTM, FTM_B. FTM_B looks up the information in its data base, and returns a

WHEROK message with the requested information to APP_B.

6. APP_2 sends a user message to APP_1. The flow of messages corresponds to the flow described in step 3, when APP_1 sent a user message to APP_2.
7. A CLOSE message is sent from APP_2 to FTM_B. This indicates that APP_2 wants to disconnect from the FTM system. FTM_B responds by returning a CLOSOK message to APP_1 and by broadcasting an appropriate KILLN message to the other FTMs in the distributed FTM system, which in this example is just FTM_A.
8. Similarly, APP_1 decides to disconnect from the distributed FTM system, so it also sends a CLOSE message to its local FTM: FTM_B. The flow of messages follows the description of the CLOSE request issued by APP_2.

CHAPTER V

FTM EVALUATION

5.1 Introduction

An evaluation of the implemented FTM system has two general purposes: First, it is necessary to *validate* the implementation. It must be shown that the FTM system processes user requests according to the user interface definitions. The various protocols must have been implemented in such a way that the distributed FTM system remains consistent in the presence of sequential and concurrent FTM system actions.

Second, the FTM *response timing* to the user requests must be quantified. Since the FTM is layered on top of the UNIX operating system, it can be expected that there will be a performance cost associated with accessing FTM services versus hard-coding the actions into user applications. A quantification of these performance costs is necessary to allow a trade-off decision between the convenience and offered fault tolerance versus the quicker response times achieved by bypassing the FTM system and formatting requests directly to the operating system.

5.2 Feature by Feature Comparison

As stated, there will always be a run-time performance cost associated with the use of FTM system services. These can be quantified and are so described in the following sections. The convenience and fault tolerance aspects of the services offered by the FTM system are not so easily quantified, but will be discussed in relation to the "raw" equivalent service offered by the operating system.

The fault tolerance services of the FTM consist of automatic user program restart and redirection of messages so that the physical node at which the message target application currently resides is transparent to the message sender application. As will be seen from the results of performance tests, program restart on an alternate node takes on the order of two seconds from the detection of the program abort. Even if a (human) operator were assigned to stare continuously at a console (or respond immediately to a bell) it is unlikely that a restart could be accomplished in this time frame. Plus, the application would have had to pick its own backup node and assure that current critical data would be available in case of an abort. All of this would have to be accomplished through some other software layered upon the operating system which, in effect, would simply duplicate the corresponding FTM service. So, these FTM services (or equivalent) are essential for time-critical application remote restarts.

Likewise, if automatic message redirection were absent, then the relocated application would need to contact all communicating applications. This means that it must know at which node each of these processes currently resides. But, since application self-relocation is assumed possible, there must be a mechanism by which an application can "look up" the current location of other applications. Again, these are duplications of FTM services. So, applications which need FTM-like remote restart and message redirection services cannot obtain them directly from UNIX; there must be some software entity to provide them. This is not to say that a system less general that the FTM could not be more efficiently implemented, but rather to propose that *some* layered software must be present to provide these services.

Even if it is assumed that applications ignore the fault tolerance offered by the FTM services, the transparent message delivery services can be very useful. There is quite a convenience in not requiring a predetermined distribution of communicating applications across the nodes of a distributed system. Message paths (called *sockets*) in UNIX are asynchronous. This means that a pair of applications, in order to create a socket connection, must each execute a distinct series of UNIX socket services calls. And, the application which is the *client* must know the physical location of the other application (the *server*). Actually, the server should also know the physical location of the client. This is because if they happen to be located on the same node, then a UNIX (single system) socket may be used rather than an INET (inter-net) socket. INET sockets work on a single system, but are less efficient than UNIX sockets. The FTM system handles all of this for the application. Only a single call to the *adv* user interface routine is required to create any number of data path connections to any number of other local and remote applications.

In the case where application relocation is not needed, however, applications can certainly bypass the FTM: they can simply all be aware of the location of each other so that messages are sent directly. Program parameters (or data files) could be used so that as each program starts up it accesses the location data and creates its message data paths (sockets) accordingly. One approach might be to produce a *Location-Connection* Code library (LCC), similar to the Intercept Code library (IC) of the FTM system. The advantage of the LCC would be that once an application has called its LCC to set up the data paths, message transfers would be direct with no overhead above the normal UNIX message transmission delays. This would be a reasonable approach for a *static* distributed application system.

Besides relocation and node-independent message delivery, the other services offered by the FTM system are FTM-specific and would be immaterial in a non-FTM environment. Copying of critical files to a backup system is vacuous in the absence of a backup system. The two informational IC user interface requests: *ftmstatus* to get a snapshot of the current status of the distributed FTM system, and *ftmwhere* to inquire about the location and backup system of some connected application, would have no use in the absence of the FTM. There is no FTM status, there is no application backup system, and each application would already know the static location of all other applications with which it might want to communicate. Finally, *ftmclose* would be unneccessary since there would be no FTM connection to close down.

5.3 Validation of Implementation

The validation of the FTM system is threefold: First, Conditional compilation switches provide extensive debugging output which has been examined to assure that the FTM system handles concurrent events in a consistent manner. Second, the FTM system has been used at Texas A&M University by several groups of programmers to help develop distributed, fault tolerant, application systems. And third, the performance evaluation tests described in later sections have been run separately and concurrently on distributed FTM systems consisting of one through 16 nodes. The FTM systems have run continuously for up to several days at a time, varying between supporting multiple applications and idle time, with no apparent ill effects from the "bombardment" of calls to all interface routines from these performance evaluation tests.

The performance evaluation tests, while designed specifically to gather timing statistics for various FTM services, also serve to validate

the interface routines. All user interface calls are utilized during this sequence of tests, and invalid handling of any of these calls would be detected.

The protocol messages occurring during "normal" system operation are also exhaustively exercised by the test programs. "Abnormal" system events, such as the failure of an application, an FTM, or an entire node, have been externally initiated. The response of the FTM system has been examined both in terms of the protocol messages generated and of the effect on the connected applications.

5.4 Baseline Performance/Reliability

This section presents the results of a series of performance tests. The physical distributed system consisted of 16 SUN 4/60 Sparc Stations running SunOs Release 4.0.3c UNIX and configured as shown in Figure 16.

The physical distributed system consists of four sets of four nodes each. Within each set, the four nodes are connected by a single ethernet. The four sets are then interconnected through a dedicated system interface unit (SIU) which forwards message packets between the sets of nodes. This gives each node the appearance of being directly connected to all 15 other nodes. There is, of course, a transmission delay if the sending node and receiving node are not within the same set. For a single packet (not more than 1024 bytes) this delay is approximately one milli-second. In any case, the following tests which require four or less nodes have been performed within a single set. The only exception is the *adv* test, where the timing is dependent on the number of nodes on which the distributed FTM system is currently running.

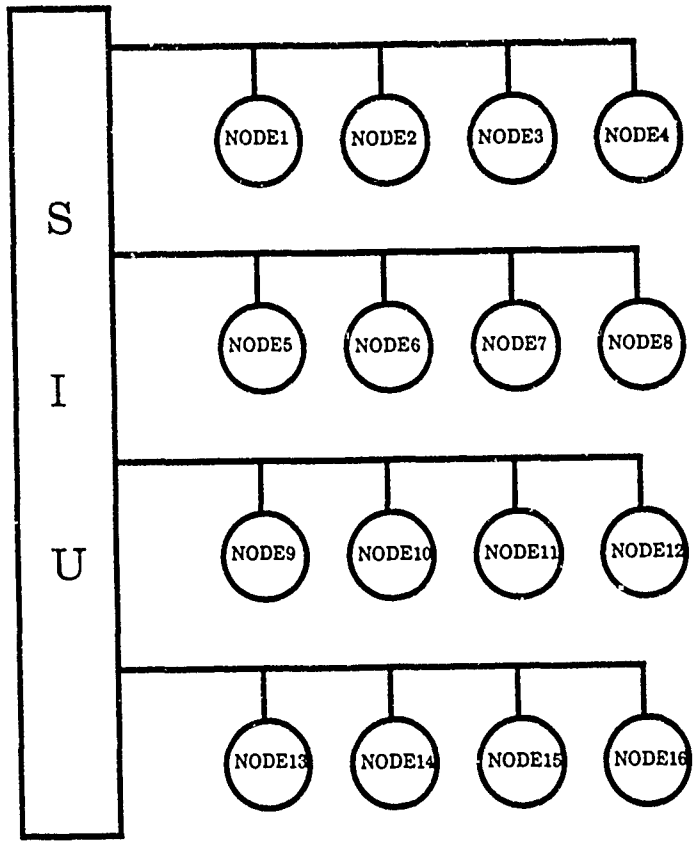

Figure 16. Distributed System for Performance Tests

It should be noted that each and every test has been repeated three times. These trials were at different times and on separate sets of nodes. Rather than attempt to extract one set of results from the three sets of test output, all three sets will be reported. While every attempt has been made to run these tests during periods of low system activity, the timing statistics do include some level of interruptions for system overhead functions.

The code for these tests appear in Appendix C. Besides showing the source for the results presented, these tests can also be used as examples of application programs which utilize FTM services.

In all of the following tables, LOOPS refers to the number of repetitions performed by each run of a particular test. All times are in milliseconds. (The resolution of the UNIX timing services employed are one milli-second).

5.4.1 Advertise a Name (*adv*)

Program *zadvclose* repeatedly advertises a name throughout the FTM system, and then disconnects from the FTM system (via a call to *ftmclose*) so that it can advertise again. Statistics are maintained for both the *adv* (Table III) and the *ftmclose* calls. Because the *adv* is sensitive to the number of nodes in the system (since the local FTM must employ a 2-phase commit protocol with all other nodes) it is tested in various configurations, from 1 system (where no commit protocol is employed) to the full 16 system configuration. Since the other FTMs will act in parallel on the commit protocol requests, little increase in delay is expected as configurations change from 2 to 16 nodes.

The average time for an *adv* jumps dramatically as the size of the distributed FTM system increases from 1 to 2 nodes, then stabilizes for further increases. In any case, an application normally advertises only once, so the 200 milli-second delay would generally be negligible and could be considered part of the normal application loading and start-up time.

5.4.2 Disconnect from FTM (*ftmclose*)

As with the *adv* call, the *ftmclose* is generally called once by an application, if at all. And, this is usually when an application is about to terminate. Table IV presents timing statistics for the *ftmclose* call. These statistics were gathered by test program *zadvclose* while it was gathering the *adv* statistics.

Table III. Timing Data: *adv*

TIME TO ADVERTISE THROUGHOUT FTM SYSTEM					
LOOPS	NODES	MAX	MIN	AVG	STDEV
5000	1	70	9	13	2
5000	1	76	9	12	2
5000	1	82	9	13	2
5000	2	250	33	190	5
5000	2	249	38	191	4
5000	2	243	33	191	4
5000	4	303	57	186	11
5000	4	305	56	187	9
5000	4	326	56	188	8
5000	8	864	94	187	23
5000	8	479	95	230	21
5000	8	494	92	209	29
5000	16	662	155	240	28
5000	16	535	152	267	31
5000	16	557	166	277	36

Table IV. Timing Data: *ftmclose*

TIME TO DISCONNECT FROM FTM SYSTEM					
LOOPS	NODES	MAX	MIN	AVG	STDEV
5000	1	68	5	7	1
5000	1	69	5	7	1
5000	1	12	5	7	1
5000	2	14	6	9	1
5000	2	66	8	9	2
5000	2	72	8	9	1
5000	4	127	11	13	3
5000	4	131	11	13	3
5000	4	88	10	13	2
5000	8	140	13	18	4
5000	8	74	17	20	2
5000	8	140	19	21	3
5000	16	168	25	28	5
5000	16	133	22	30	6
5000	16	148	27	31	4

5.4.3 Obtain FTM Status Information (*ftmstatus*)

Test program *zstatus* repeatedly calls the *ftmstatus* user interface routine. This call returns information about the current status of the distributed FTM system (Table V). Because the local FTM retrieves the status information from its internal tables, the timing is not dependent on the number of nodes in the FTM system. Currently, the status information is returned in two integers (64 bits).

Table V. Timing Data: *ftmstatus*

TIME TO OBTAIN FTM STATUS				
LOOPS	MAX	MIN	AVG	STDEV
10000	87	6	9	2
10000	101	7	9	2
10000	58	6	9	1

5.4.4 Obtain Node of an Application (*ftmwhere*)

Test program *zwhere* repeatedly calls the *ftmwhere* user interface routine. This call returns the current and backup nodes for a specific application (Table VI). As with the *ftmstatus* call, the local FTM obtains the information from its internal tables. Thus the timing is not dependent on the number of nodes in the FTM system. The information returned is based on the advertised name passed as a parameter to the call.

Program *zdummy*, running on some remote node, serves as a dummy target application about which location information is requested. Program *zdummy* does nothing more than advertise a name, and then receive (and delete) any messages which may be sent to it. It is a passive partner to any active statistic gathering program.

Table VI. Timing Data: *ftmwhere*

TIME TO FIND APPLICATION NODE DATA				
LOOPS	MAX	MIN	AVG	STDEV
10000	86	7	10	2
10000	107	7	10	2
10000	84	7	10	2

5.4.5 Send a User Message (*to*)

Since sending messages is the FTM supported activity likely to comprise the overwhelming majority of FTM requests by a typical application, timing statistics for the user interface routine which sends messages (*to*) will be more detailed. The data covers a range of user message sizes, and is reported for the case where the sender application and target application reside on the same physical node, and then is repeated for the case where they reside on different physical nodes. In addition, data is reported for a pair of applications communicating directly rather than through the FTM system. Again, there are separate statistics for the case where the application pair reside on the same physical node, versus distributed applications.

Test program *zto* sends a number of messages of a given size to program *zdummy*. Program *zdummy* simply discards the message. The number of messages and message size are parameters to *zto*.

The data for the case where an application sends messages to a target application which happens to reside on the same physical node is given in Table VII. The time reported for each message represents the total time of the following individual steps:

1. The sender application calls *to* which results in the message being sent to the local FTM over a UNIX socket.

2. The local FTM discovers that the target application is on the same node, so it forwards the user message to the target application over a UNIX socket. The local FTM then sends a UNIX *signal* (interrupt) to the target application.
3. The Intercept Code within the target application obtains control (due to the signal) and receives the user message. The message is queued so that the target application can read it (with a *from* user interface call). The target application IC then constructs a response message and sends it to the local FTM over a UNIX socket.
4. The local FTM receives the response message from the target IC and forwards it to the sender application over a UNIX socket. Assuming the response message indicates that the user message was delivered correctly, the IC in the sender application returns from the *to* call with a "success" status.

The time reported for each *to* is actually the time for a total of four message hops, all over UNIX sockets. The entire message is sent over each hop, including the return path, so the response message can be compared with the original user message.

As expected, the delay increases with message length. But, the increase is much less than proportional, since there is an "almost fixed" delay involved both in FTM processing and UNIX processing for each message. Notice that the FTM system allows zero length user messages. In this case, the target application will be able to read the message, but of course the length will be returned as zero. This could be useful, for example, as an "I'm still alive" message where actual data is not needed. The FTM system will provide the target application with the advertised name

of the sender.

Table VII. Timing Data: *to* (Single Node)

TIME TO SEND DATA (SINGLE NODE)					
LOOPS	SIZE	MAX	MIN	AVG	STDEV
5000	0	90	15	19	2
5000	0	124	17	19	2
5000	0	91	17	19	2
5000	100	93	21	21	2
5000	100	122	21	22	3
5000	100	97	21	22	2
5000	512	95	22	23	2
5000	512	126	22	24	3
5000	512	95	22	23	2
5000	1024	98	25	26	2
5000	1024	128	25	26	3
5000	1024	97	25	26	3
5000	2048	104	31	32	3
5000	2048	132	31	32	3
5000	2048	106	31	32	3

The data for the case where an application sends messages to a target application which happens to reside on a different physical node is presented in Table VIII. This increases the number of steps necessary to accomplish the transmission:

1. The sender application calls *to* which results in the message being sent to the local FTM over a UNIX socket.
2. The local FTM discovers that the target application is on a different node, so it forwards the user message to the FTM on the target node over an INET (internet) socket.
3. The FTM on the target node forwards the message to the target application over a UNIX socket. The target FTM then sends a UNIX signal to the target application.
4. The Intercept Code within the target application obtains

control (due to the signal) and receives the user message. The message is queued so that the target application can read it (with a *from* user interface call). The target application IC then constructs a response message and sends it to its local FTM (the target FTM) over a UNIX socket.

5. The target FTM receives the response message from the target IC and forwards it on to the sender FTM over an INET socket.
6. The sender FTM receives the response message from the target FTM and forwards it on to the sender IC over a UNIX socket. Assuming the response message indicates that the user message was delivered correctly, the IC in the sender application returns from the *to* call with a "success" status.

Note that this sequence of steps is very similar to that of the case where the sender and target applications were on the same node, except that now there are six message hops: the same four UNIX socket message transmissions and an additional two INET socket transmissions between the two FTMs. So, the time reported for each *to* is actually the time for a total of six message hops, four UNIX and two INET. Again, the entire message is sent over each hop; including the return path, so that the response message can be compared with the original user message.

A brief comparison of Tables VII and VIII shows that the delay time in sending a user message through the FTM system is approximately 50% higher if the sender and target applications are running at different nodes.

Whether or not these delays are acceptable is application dependent. As an aid to comparison, data derived from "raw" UNIX message

communication is also presented. In a fashion similar to the way that programs *zto* and *zdummy* act together to accumulate and report timing statistics through the FTM, program *zzto_un* acts with program *zzdummy_un* to accumulate and report timing statistics directly over UNIX sockets when sender and target both reside on a single node. Similarly, program *zzto_in* acts with program *zzdummy_in* to accumulate and report timing statistics over direct INET sockets when sender and target reside on separate nodes.

Table VIII. Timing Data: *to* (Separate Nodes)

TIME TO SEND DATA (SEPARATE NODES)					
LOOPS	SIZE	MAX	MIN	AVG	STDEV
5000	0	103	26	28	3
5000	0	116	25	29	3
5000	0	101	25	28	3
5000	100	110	33	35	3
5000	100	123	33	35	4
5000	100	99	32	35	3
5000	512	113	35	37	3
5000	512	127	34	37	4
5000	512	105	32	35	3
5000	1024	114	37	41	3
5000	1024	130	38	41	4
5000	1024	111	38	41	4
5000	2048	128	50	52	4
5000	2048	142	50	52	4
5000	2048	121	48	52	4

As can be seen from a cursory comparison of Tables VII and IX, the delay involved in sending a message through the FTM is on the order of 10 to 20 times longer than the delay involved in using a direct UNIX socket. The FTM system overhead could be cut by eliminating the full user data from being transmitted along the return path. This would delete a level of fault detection from the FTM system. The speedup would be

minimal and the direct socket would remain an order of magnitude faster.

Table IX. Timing Data: UNIX socket (Single Node)

TIME TO SEND DATA DIRECTLY (SINGLE NODE)					
LOOPS	SIZE	MAX	MIN	AVG	STDEV
5000	100	4	1	1	0
5000	100	5	1	1	0
5000	100	3	1	1	0
5000	512	4	1	1	0
5000	512	4	1	1	0
5000	512	4	1	1	0
5000	1024	4	1	1	0
5000	1024	97	1	1	1
5000	1024	4	1	1	0
5000	2048	4	1	2	1
5000	2048	4	1	2	1
5000	2048	4	1	2	1

Table IX also displays an exceptional condition which should be mentioned. It can be seen that for one of the test runs, the maximum response time (MAX in the table) is 97 milli-seconds, while the MAX values for the other 11 runs range from three to five milli-seconds. And, Table IX represents a test run in which the FTM system is not present (direct UNIX socket communication is being timed). With or without the presence of the FTM system, the response time depends on the current UNIX activity level. Despite attempts to perform these tests during periods of minimal system activity, these timing spikes can occur depending on the load of the system. The number of iterations per test (LOOPS in the table) is set high enough to minimize the effect of such anomalies. This particular anomaly can be seen to have had no effect on the average delay time for the set of 5000 iterations.

The timing data for a direct connection over an INET socket between applications on separate nodes is given in Table X.

Table X. Timing Data: INET socket (Separate Nodes)

TIME TO SEND DATA DIRECTLY (SEPARATE NODES)					
LOOPS	SIZE	MAX	MIN	AVG	STDEV
5000	100	96	2	3	1
5000	100	109	2	3	2
5000	100	106	2	3	2
5000	512	99	3	4	1
5000	512	110	3	4	2
5000	512	105	3	4	2
5000	1024	103	4	5	2
5000	1024	111	4	5	2
5000	1024	85	4	5	1
5000	2048	102	5	6	2
5000	2048	110	5	6	2
5000	2048	106	5	6	2

A quick comparison of the average delay times as shown in Tables VIII and X show that for communicating applications on separate nodes, the FTM time delay penalty is slightly less than an order of magnitude. Again, the additional delay versus convenience and fault tolerance trade-offs must be made on an application by application basis.

5.4.6 Critical File Backup (*copycf*)

Currently, the FTM system keeps track of backup systems. A standard UNIX network utility (*rcp* – remote file copy) is invoked by the FTM system to perform the actual data copy. The timing for a remote copy to the backup system is therefore dependent only on the file size and the current UNIX and network activity level.

In section 5.5, timing statistics for remote restarts will be presented. These will include the *copycf* calls necessary to provide critical file information at the backup nodes.

5.4.7 Read a User Message (*from*)

The *from* call is the only IC user interface routine which does not result in a communication protocol exchange with the local FTM. Recall that when a message is sent (with the *to* call) the message is delivered to the IC of the target application. The IC gains control as a result of an interrupt from the target FTM, at which time the target IC queues the message, constructs an acknowledgment, and then returns control to the interrupted application.

This means that for a *from* call, the IC requires only a scan of its queued messages. If a message matching the request is found, it is returned to the calling application. Otherwise, the application is immediately informed that no appropriate message has yet been queued. In either case, no communication is involved, and thus the time would be insignificant compared to all other user interface calls.

5.5 Automatic Relocation of Failing Applications

This is the only FTM service that is not performed as the direct result of a call to a user interface routine. Rather, the FTM system monitors connected applications. Any unrecoverable problem with the communication link between an FTM and one of its connected applications invokes the restart/relocation service (An application is *connected* to the FTM system if it has made a successful call to *adv* and has not made a subsequent call to *ftmclose*).

An FTM which has detected the failure of a connected application informs all remote FTMs. As a result, all FTMs delete the advertised name from their internal data tables (structure *namtable*). The backup FTM, after deleting the old advertised name, then attempts to perform a restart. The *ftmbkup.lis* file is examined and, if the advertised name is found, the corresponding path is used in the attempt of the restart. Any application which desires to be eligible for restart is responsible for making periodic calls to user interface routine *copycf* if critical data is required to be present at the backup node.

5.5.1 Relocation Statistics

Program *zflop* is used to generate timing statistics on program relocation/restarts. Because there is no central clock, *zflop* cycles continuously through the nodes of the distributed FTM system but calculates the statistics only as it is cycling through the node on which it originally started. It then assumes that each node transition in the current cycle took equal time. This does not affect the average time reported for the multiple cycle run, but the standard deviation will admittedly be somewhat understated.

Since the FTM system itself dynamically chooses backup nodes based on the current distribution of connected applications, program *zflop* cannot choose its own backup node as the next node in a predetermined cycle. Only if there is an equal number of connected applications at each node in the FTM system at each *zflop* startup will *zflop* cycle correctly. Or, if there are only two nodes in the current FTM system then *zflop* will be forced to cycle between the two nodes (i.e.: *flop* back and forth) regardless of connected application distribution. This has the additional advantage that the understatement of standard deviation is minimized.

The operation of program *zflop* is straightforward. After startup, *zflop* attempts to access its critical file. If the file is unavailable, *zflop* assumes that this is the initial start so it processes the command line parameters to obtain the LOOPS (number of restarts required before final termination) and the number of nodes to expect in one cycle. This information, the current loop count (0 for the initial start), running totals for eventual statistics calculations (again, 0 for the initial start), and other such information is written to a critical file (about 65 bytes). Next, *copycf* is called to have the FTM system copy this file to the backup system. Program *zflop* next exits without calling *ftmclose*, so that the FTM system will consider the termination as abnormal, and will restart *zflop* on the backup system

When *zflop* is restarted on a node other than its initial node, it realizes that is is within a cycle (by reading the information in its critical file) so it simply updates its current loop count, rewrites the critical file, calls *copycf* to copy the critical file to its current backup system, and exits without calling *ftmclose*.

When *zflop* is restarted on its initial node, which it discovers by reading its critical file and calculating that its current loop count is a multiple of the number of nodes in a cycle, it performs timing analysis for the current cycle and updates the running totals before rewriting its critical file. Again, *copycf* is called, followed by an exit without calling *ftmclose*.

Eventually, *zflop* will complete the number of loops passed as an original parameter and carried along from critical file to critical file. At this point, timing statistics are generated (from the running totals in the critical file) and output to the critical file. There is now no need to call *copycf* since *zflop* no longer desires to be restarted. So, *zflop* calls *ftmclose*

to disconnect from the FTM system. Finally, *zflop* exits, and is not restarted by the FTM system because this termination is not considered abnormal.

Table XI. Timing Data: Program Restart/Relocation

TIME TO *copycf* AND RESTART/RELOCATE				
LOOPS	MAX	MIN	AVG	STDEV
1000	3.999	1.494	1.932	0.302
1000	4.213	1.357	1.819	0.293
1000	2.700	1.477	1.621	0.142

The times reported in Table XI are in seconds (all previous tables in this chapter report time in milli-seconds). Each reported time includes a backup file copy (*copycf*) of about 65 bytes, the detection of the *zflop* program exit by its local FTM, the dissemination of this information by the local FTM to the remote FTMs, and finally the program loading and restart by the backup FTM at the backup node.

5.6 Timing Statistics Summary

Figures 17 and 18 present a summary of the timing statistics for the various FTM system calls. The detailed statistics can be found in Tables III – X.

Figure 17 summarizes the observed time samples for the *adv* (both multiple system and single system cases), the *ftmclose*, the *ftmstatus*, and the *ftmwhere* FTM user interface calls. Recall that the *adv* and *ftmclose* routines will each be called only once in a typical application. The *ftmstatus* and *ftmwhere* routines will probably not be called at all in most applications.

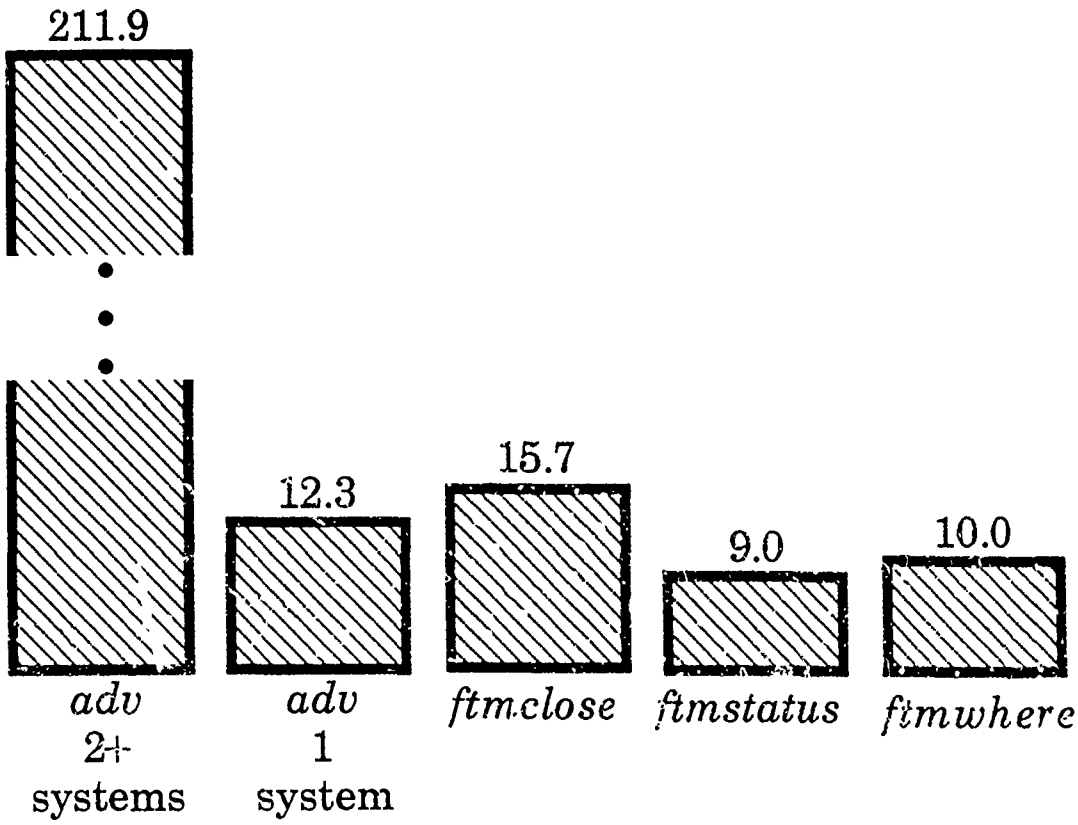

Figure 17. Summary of FTM Call Times

The *to* user interface call, which delivers user messages to other local and remote applications, is likely to comprise the vast majority of FTM user interface calls executed by a typical application. The values in Figure 17 are averages taken from Tables III - VI.

Figure 18 is a graph which displays the user message transmission times for varying message sizes. Four cases are graphed: *to* where the sender and target applications are on the same node, *to* where the sender and target applications are on remote nodes, direct (non-FTM) message transmission over UNIX sockets between sender and target on the same node, and direct (non-FTM) message transmission over INET sockets where the sender and target applications are on remote nodes. The data

points for the graph of Figure 18 are taken from Tables VII - X.

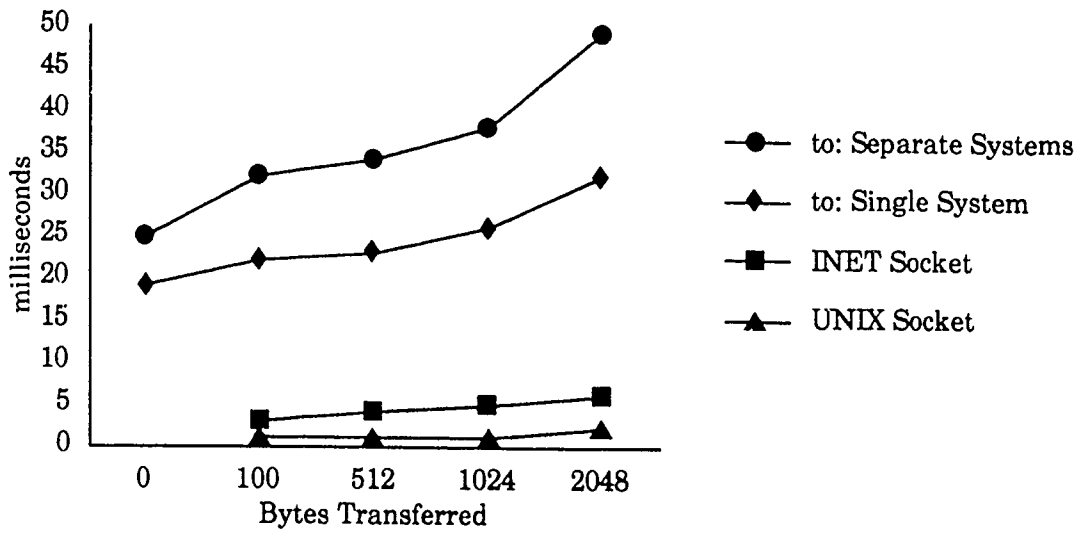

Figure 18. Summary of Message Transmission Times

5.7 Reliability Matters

Any attempt to quantify the enhanced reliability offered by the fault tolerance services of the FTM must be based on the topology of the particular network being examined. Consider the simplest possible network: an n node network of similar processors, with l replicated links. Each link is a single path connecting all the nodes.

For the simple network of Figure 19, it is easy to calculate the system reliability given the individual reliabilities of the nodes and links. For this discussion, *reliability* will be interpreted as the *probability* of correct operation for some given unit of time.

Let: R_n be the *reliability* of each node

R_l be the *reliability* of each link

n be the number of nodes

l be the number of links

k be the number of programs in an application group

Assuming each program must execute on a separate processor, we can calculate the reliability of the system both in the case without an FTM (where the particular k nodes and at least 1 link must remain up) and in the case with an FTM (where any k nodes and at least 1 link must remain up):

Without FTM:

$$R_n{}^k \cdot \left[1-(1-R_l)^l\right]$$

With FTM:

$$\left[\sum_{i=k}^{n}\binom{n}{i}R_n{}^i \cdot (1-R_n)^{n-i}\right] \cdot \left[1-(1-R_l)^l\right]$$

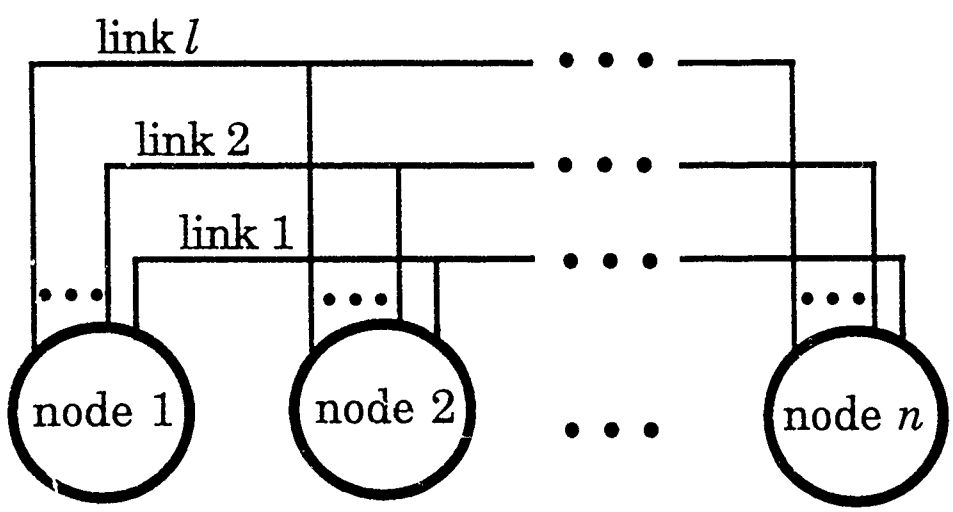

Figure 19. Simple Network Topology

Ignoring the reliability of the links (since it is the same in either case), Figure 20 graphs the reliability (both with and without the FTM

system) for a 16 node system where the application group requires 5 processors. A simplifying assumption made for Figure 20 is that the FTM system is fully reliable, or at least it will continue to operate, on 5 or more nodes.

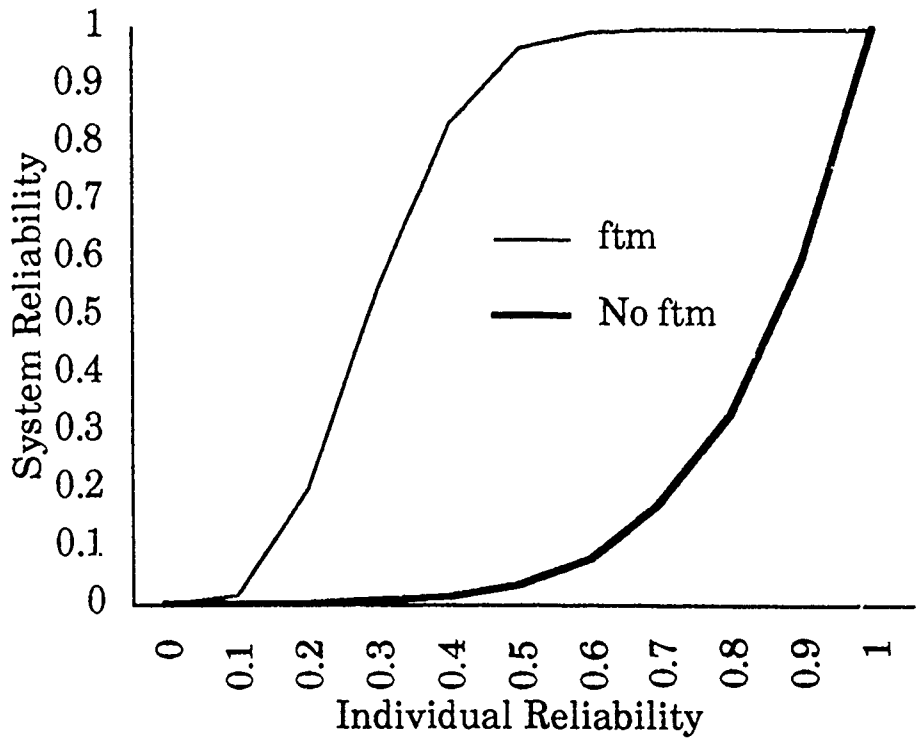

Figure 20. System Reliability on 5 of 16 Nodes

A more realistic network topology to take advantage of the FTM fault tolerance services would be one such as planar-2 (see Figure 10). In this case the analysis of fault tolerance levels becomes more complex. For the planar-2 topology, Turner [88] analyzed failure rates for systems with relocatable applications.

Included in his analysis is the code and sample output from a Monte Carlo simulation package provided by this author for validation of

Turner's results in certain cases. The simulation package analyzes an N x M planar-2 network with individually definable reliabilities for each node and link. The reliability of the network is investigated in terms of the maintenance of a connected component containing at least as many nodes as required by some given application system.

The FTM system facilitates the implementation of software *design diversity* into connected applications. As previously discussed, software design diversity involves the independent creation and validation of several software modules, each designed to perform identical processing. In this case "identical processing" means that any given input would result in identical output from either of the several software modules. If the FTM system detects a failure in a connected application, it refers to the *ftmbkup.lis* file in order to obtain the path to the file containing the executable. This file need not contain the original executable; it may contain another software module meant to replace the original.

CHAPTER VI

CONCLUSIONS AND RECOMMENDATIONS

6.1 Introduction

The primary objective of this research was to design and develop an approach to fault tolerance in a loosely coupled environment. Among the restraints was the requirement that applications which utilize the fault tolerance services must run along with applications which do not. And, the applications which do not utilize these services should be in no way affected. The services need not be transparent to the requesting applications, but should be convenient.

A literature survey was conducted into recent approaches to fault tolerance, with emphasis on network environments. A variety of implementations were studied, including distributed operating systems and distributed programming languages. From this background, an approach was chosen: to attempt to *layer* a fault tolerant environment on top of a standard operating system. Protocols were developed to implement this fault tolerant environment. The UNIX operating system was chosen for an implementation platform since it is the operating system whose current availability covers the widest selection of hardware platforms.

6.2 Conclusions

The successful implementation of an FTM system employing the protocols developed demonstrates the feasibility of the approach chosen. Several projects at Texas A&M University are under development which use the FTM environment as the sole platform to handle message transmission between applications. For some projects, including

applications designed to support remote autonomous distributed systems, the failure detection and process relocation FTM services are the required feature. For others, the convenient node-independent FTM message delivery system is used even though FTM fault tolerance services are not required.

The usefulness of the FTM approach to layered fault tolerance is further evidenced by continuing research into the area. A next generation Fault Tolerant Monitor Protocol (FTMP) is being developed here at Texas A&M. The FTMP will support all of the services of the current FTM system, along with enhancements. Requests and suggestions from application developers using the current FTM system will be the impetus behind many of these extensions.

6.3 Recommendations

The implemented FTM system has been used by several groups of programmers at Texas A&M University for over a year. Based on this usage, a number of shortcomings have been noticed. Many of these recommendations are being incorporated into the next version, currently under development.

The FTM system currently does not recognize the presence of a cooperating sub-group of applications among the group of currently connected applications. Each application is treated as independent, so any application may communicate with any other. Since several suggested enhancements to the FTM system will involve control over certain applications by certain other applications, it is desirable that the FTM system be able to divide the applications into separate groups.

This concept could be defined by an abstraction called a *project*. An application could issue a *define_project* call, which would identify the applications in the group. Each of these applications would subsequently issue a *join_project* call.

This concept of *projects* would address several other deficiencies which have been noted in the current implementation of the FTM system. While failing applications can currently be restarted and relocated upon failure, they cannot be initially started up. Since *join_project* would identify the applications in the group, the FTM system could easily perform these initial executions of applications.

The *define_project* (and *join_project*) calls could also provide the FTM system with other application-specific information. This might include resources required by the application which are not available at every node. This information would be essential if the FTM is to choose a useful backup system for the application. Similarly, if all nodes were not binary compatible then information as to the paths of several different executables for an application could be provided to the FTM system so that it could restart the application correctly.

The current implementation of the FTM system does no routing; this is left to lower layers of the internet protocols (TCP/IP on the implementation network). While this is sufficient for simple networks, the standard UNIX router will fail in the presence of cycles. Examine Figure 10 of Chapter II (Planar-2 Network Topology). The advantages of this topology in the face of multiple node and link failures, over that of a duplicated link connecting all nodes, is obvious.

While certain routers are available which will handle cycles, the FTM system can increase performance by maintaining its own routing tables based on its knowledge of current network topology. This would include the current "up or down" status of each node and link.

Each FTM is responsible for checking its connected applications. An application which is found to have terminated (or at least has been found to be unreachable) is disconnected from the FTM system by its local FTM, and that information is propagated throughout the FTM system. This may result in a restart on a remote system. Because it is difficult for an FTM to detect "aberrant" program behavior from an application, this task will most likely be left to other applications within the *project*. The FTM system could, however, provide support for the abort of an application which has been identified by other applications within its *project* as exhibiting aberrant behavior.

Another area for future enhancement would be the development of *health managers*. While applications within a *project* are expected to check each other for abnormal behavior, applications should not be required (or able) to check and control the FTM system itself. Currently, if any FTM detects that it cannot communicate with a remote FTM, it immediately declares that remote FTM to be down and it so informs all other FTMs. A more sophisticated health management among the FTMs is required.

Two approaches are suggested: First, each FTM could contain routines to check on remote FTMs. A consensus mechanism could then be employed to isolate failed (or aberrant) FTMs from the active distributed FTM system. Second, a number of separate processes could be employed. These processes could have as their sole purpose the "management" of the FTM system.

A mechanism to allow cooperating applications to coordinate actions would be useful. Currently, this can be accomplished only through an interchange of messages. A *distributed object service* could allow for more elegant and efficient distributed cooperation. A *distributed object* is a data item owned by a single application. The owning application could create the named object and could update it as required. All other applications would be authorized only to read the object.

This *distributed object service* could be added to the FTM system as a generalization of the current *advertise* service, where an application requests the FTM system to make a unique name known throughout the distributed system. The same 2-phase commit protocol could be used to assure that a *distributed object* name is unique throughout the entire FTM system.

Applications currently must *poll* for incoming messages. An application makes a call to the *from* interface routine in order to check for the presence of messages from other applications. If no messages are present, the *from* returns immediately with an appropriate status code. The option of a *blocking from*, or an *interrupt* of the application upon message receipt, could allow the application to employ the most appropriate of the three mechanisms. In general, operating systems (including UNIX) allow all three of these approaches for input processing.

Each application is now allowed to advertise a single name. All communicating applications send messages to this name. For convenience in prioritizing incoming messages, the application should be able to define multiple message gathering facilities (*named ports*). This would be especially useful if a *blocking from* and message receipt *interrupt* service, as described above, were implemented. Separate *named ports* within an

application could handle incoming messages as appropriate for distinct message priority levels.

Critical files are copied in their entirety to an application's backup system upon request to the FTM system. While this is appropriate for small files containing basic application restart information, it is not appropriate for larger local files, such as *log* files, which provide a type of audit trail of application actions. For these files, it would be preferable to initially copy the file (if it exists) to the backup system at application startup, and then update the remote backup copy of the file as the application updates the local primary copy.

The FTM system employs a *semi-dynamic* application backup system. As an application starts, the FTM system dynamically chooses a backup system based on the current distribution of workload amongst the nodes in the FTM system. However, once chosen this backup becomes static until the application terminates. Changing workload distribution should cause updated backup system definitions. In addition, when a node fails then any application which has that node as a backup should dynamically be assigned a new backup system. Very critical applications should also be able to request multiple backup systems.

Applications are being written to run under the FTM system which are sensor and control oriented. It is common for sophisticated sensors and controls to be able to connect directly to a communication link, such as Ethernet, and to contain support in read-only memory (ROM) for communications protocols, such as TCP/IP. If application programs running on a distributed network are communicating through the FTM system, then some of the FTM services (such as node-independent message delivery) could be of use to the sensors and controls also. However, each

sensor or control would not contain a full operating system and therefore could not run a full FTM.

A subset of the FTM system protocols could be implemented as a *trivial* FTM. This *trivial* FTM running on a sensor or control would allow it to communicate through the FTM system to an FTM on a full connected node. This FTM would then handle the required FTM services for the *trivial* FTM.

These suggestions all *extend* the collection of services provided by the FTM system. This large variety of additional services could be offered without altering the fundamental concept of the FTM system as a fault tolerance *layer* between the application and the standard operating system.

REFERENCES

1. Ahmad, S.H., A Simple Technique for Computing Network Reliability. *IEEE Transactions on Reliability R-31*, 1 (April 1982) 41-44.

2. Ahmad, S.H., and Jamil, T.M., A Modified Technique for Computing Network Reliability. *IEEE Transactions on Reliability R-36*, 5 (December 1987) 554-556.

3. Ammann, P.E., and Knight, J.C., Data Diversity: An Approach to Software Fault Tolerance, *IEEE Transactions on Computers 37*, 4 (April 1988) 418-425.

4. Andrews, G.R., Synchronizing Resources. *ACM Transactions on Programming Languages and Systems 3*, 4 (October, 1981) 405-430.

5. Andrews, G.R., Olsson, R.A., Coffin, M., Elshoff, I., Nilson, K., Purdin, T., and Townsend, G., An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems 10*, 1 (January, 1988) 51-86.

6. Avizienis, A., et al., The STAR (Self Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design. *IEEE Transactions on Computers C-20*, 11 (November 1971) 1312-1321.

7. Avizienis, A., and Kelly, P.J., Fault Tolerance by Design Diversity: Concepts and Experiments. *Computer 17*, 8 (August 1984) 67-80.

8. Avizienis, A., and Rennels, D.A., The Evolution of Fault Tolerant Computing at the Jet Propulsion Laboratory & at UCLA 1960-1986. Technical Report CSD-870022, UCLA Computer Science Department, 1987.

9. Babaoglu, Ozalp, On the Reliability of Consensus-Based Fault-Tolerant Distributed Computing Systems. *ACM Transactions on Computer Systems 5*, 3 (November, 1987) 394-416.

10. Bendell, T., A Classification System for Reliability Models. *IEEE Transactions on Reliability R-33*, 2 (June 1984) 160-164.

11. Berg, M., and Koren, Israel, On Switching Policies for Modular Redundancy Fault Tolerant Computing Systems. *IEEE Transactions on Computers C-36*, 9 (September 1987) 1052-1062.

12. Birman, K.P., and Joseph, T.A., Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems 5*, 1 (February 1987) 47-76.

13. Black, D.L., Golub, D.B., Hauth, K., Tevanian, A., and Sanzi, R., The MACH Exception Handling Facility. Technical Report CMU-CS-88-129, Carnegie-Mellon University, 1988. *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, (5-8 May 1988) 45-56.

14. Blaum, M., The Reliability of Single-Error Protected Computer Memories. Technical Report RJ5318, IBM Research Division, Yorktown Heights, New York, 1986.

15. Bolmarcich, A.S., An Introduction to MACH/EPEX. Technical Report RC14369, IBM Research Division, Yorktown Heights, New York, 1989.

16. Chattergy, R., and Pooch, U.W., A Distributed Function Computer for Time Sharing. *British Computer Journal 22*, 1 (February, 1979) 27-34.

17. Chillarege, R., and Bowen, N.S., Understanding Large System Failures – A Fault Injection Experiment. Technical Report RC14233, IBM Research Division, Yorktown Heights, New York, 1988.

18. Clark, D.D., The Design Philosophy of the DARPA Internet Protocols. Presented during SIGCOMM 88 Symposium on Communications Architectures & Protocols, Stanford, California (16-19 August 1988). *Computer Communications Review 18*, 4 (August 1988) 106-114.

10. Bendell, T., A Classification System for Reliability Models. *IEEE Transactions on Reliability R-33*, 2 (June 1984) 160-164.

11. Berg, M., and Koren, Israel, On Switching Policies for Modular Redundancy Fault Tolerant Computing Systems. *IEEE Transactions on Computers C-36*, 9 (September 1987) 1052-1062.

12. Birman, K.P., and Joseph, T.A., Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems 5*, 1 (February 1987) 47-76.

13. Black, D.L., Golub, D.B., Hauth, K., Tevanian, A., and Sanzi, R., The MACH Exception Handling Facility. Technical Report CMU-CS-88-129, Carnegie-Mellon University, 1988. *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, (5-8 May 1988) 45-56.

14. Blaum, M., The Reliability of Single-Error Protected Computer Memories. Technical Report RJ5318, IBM Research Division, Yorktown Heights, New York, 1986.

15. Bolmarcich, A.S., An Introduction to MACH/EPEX. Technical Report RC14369, IBM Research Division, Yorktown Heights, New York, 1989.

16. Chattergy, R., and Pooch, U.W., A Distributed Function Computer for Time Sharing. *British Computer Journal 22*, 1 (February, 1979) 27-34.

17. Chillarege, R., and Bowen, N.S., Understanding Large System Failures – A Fault Injection Experiment. Technical Report RC14233, IBM Research Division, Yorktown Heights, New York, 1988.

18. Clark, D.D., The Design Philosophy of the DARPA Internet Protocols. Presented during SIGCOMM 88 Symposium on Communications Architectures & Protocols, Stanford, California (16-19 August 1988). *Computer Communications Review 18*, 4 (August 1988) 106-114.

19. Cook, R.P., *MOD - A Language for Distributed Programming. *IEEE Transactions on Software Engineering SE-6*, 6 (November 1980) 563-571.

20. Cristian, F., Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. Technical Report RJ5964, IBM Almaden Research Center, San Jose, California, 1988.

21. Dolev, D., Feitelson, D.G., and Sompolinsky, H., Majority Voting in the Presence of Random Voters. Technical Report RJ6538, IBM Research Division, Yorktown Heights, New York, 1988.

22. Dolev, D., Halpern, J.Y., Simons, B., and Strong, R., Dynamic Fault-Tolerant Clock Synchronization. Technical Report RJ6722, IBM Research Division, Yorktown Heights, New York, 1989.

23. Dolev, D., Halpern, J., Simons, B., and Strong, R., A New Look at Fault Tolerant Network Routing. Technical Report RJ4238, IBM San Jose Research Laboratory, San Jose, California, 1984.

24. Everhart, C.F., Making Robust Programs. Technical Report CMU-CS-85-170, Carnegie-Mellon University, 1985.

25. Feder, T., Reliable Computation by Networks in the Presence of Noise. Technical Report RJ6310, IBM Research Division, Yorktown Heights, New York, 1988.

26. Feldman, J.A., High Level Programming for Distributed Programming. *Communications of the ACM 22*, 6 (June 1979) 353-368.

27. Frost, D.K., and Poole, K.F., A Method for Predicting VLSI-Device Reliability Using Series Models for Failure Mechanisms, *IEEE Transactions on Reliability R-36*, 2 (June 1987) 234-242.

28. Garey, M.R., and Johnson, D.S., *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York (1979) 211.

29. Goyal, A., and Lavenberg, S.S., Modeling and Analysis of Computer System Availability. Technical Report RC12458, IBM Research Center, Yorktown Heights, New York, 1986.

30. Goyal, A., and Tantawi, A.N., A Measure of Guaranteed Availability and its Numerical Evaluation. *IEEE Transactions on Computers 37*, 1 (January 1988) 25-32.

31. Goyal, A., Nicola, V.F., Tantawi, A.N., and Trivedi, K.S., Reliability of Systems with Limited Repairs. *IEEE Transactions on Reliability R-36*, 2 (June 1987) 202-207.

32. Griefer, A., and Strong, R., DCF: Distributed Communications with Fault Tolerance. Technical Report RJ6361, IBM Research Division, Yorktown Heights, New York, 1988.

33. Hecht, H., and Dussault, H., Correlated Failures in Fault-Tolerant Computers. *IEEE Transactions on Reliability R-36*, 2 (June 1987) 171-175.

34. Herlihy, M., Concurrency vs. Availability: Atomicity Mechanisms for Replicated Data, Technical Report CMU-CS-87-172, Carnegie-Mellon University, 1987.

35. Hopkins, A., Basil Smith, T., and Lala, J., FTMP – A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft. *Proceedings of the IEEE 66*, 10 (October 1978) 1221-1239.

36. Hseuh, M.C., Iyer, R.K., and Trivedi, K.S., Performability Modeling Based on Real Data: A Case Study. *IEEE Transactions on Computers 37*, 4 (April 1988) 478-484.

37. Ihara, K., and Mori, K., Autonomous Decentralized Computer Control Systems. *Computer 17*, 8 (August 1984) 57-66.

38. Irland, E.A., Assuring Quality and Reliability of Complex Electronic Systems: Hardware and Software. *Proceedings of the IEEE 76*, 1 (January 1988) 5-18.

39. Ishida, Y., Tokumara, H., and Adachi, N., Diagnosability and Distinguishability Analysis and its Applications. *IEEE Transactions on Reliability R-36*, 5 (December 1987) 531-538.

40. Johnson, D., The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems. *Computer 17*, 8 (August 1984) 40-48.

41. Kane, V.R., Real-Time Kernel, Real-Time Applications. *ESD: THE Electronic System Design Magazine 18*, 8 (August 1988) 55-61.

42. Katsuki, D., et al., Pluribus - An Operational Fault- Tolerant Multiprocessor. *Proceedings of the IEEE 66*, 10 (October 1978) 1146-1159.

43. King, R.P., Halim, N., Garcia-Molina, H., and Polyzois, C.A., Management of a Remote Backup Copy for Disaster Recovery. Technical Report RC14291, IBM Research Division, Yorktown Heights, New York 1988.

44. Kirschen, D., An Overview of the Mach Operating System. *IEEE TCOS Newsletter 3*, 2 (June 1989) 5-7.

45. Kohler, W.H., A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *Computing Surveys 13*, 2 (June 1981) 149-183.

46. Koo, R., and Toueg, S., Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering SE-13*, 1 (January 1987) 23-31.

47. Kuhl, J.G., and Reddy, S.M., Fault-Tolerance Considerations in Large, Multiple-Processor Systems. *Computer 19*, 3 (March 1986) 55-67.

48. Lee, Y., and Shin, K.G., Optimal Design and Use of Retry in Fault-Tolerant Computer Systems. *Journal of the ACM 35*, 1 (January 1988) 45-69.

49. Lehr, T., Black, D., Segall, Z., and Vrsalovic, D., MKM: Mach Kernel Monitor Description, Examples and Measurements. Technical Report CMU-CS-89-131, Carnegie-Mellon University, 1989.

50. Liskov, B., and Scheifler, R., Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *Proceedings of the 9th Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, (25-27 January 1982) 7-19.

51. Mahmood, A., and McCluskey, E.J., Concurrent Error Detection Using Watchdog Processors - A Survey. *IEEE Transactions on Computers 37*, 2 (February 1988) 160-173.

52. Matloff, N.S., A Multiple-Disk System for Both Fault Tolerance and Improved Performance. *IEEE Transactions on Reliability R-36*, 2 (June 1987) 199-201.

53. McDonald, D.B., CMU Common Lisp User's Manual, MACH/IBM RT PC Edition, Technical Report CMU-CS-87-156, Carnegie-Mellon University, 1987.

54. McQuillan, J.M., Falk, G., and Richer, I., A Review of the Development and Performance of the ARPANET Routing Algorithm. *IEEE Transactions on Communications COM-26*, 12 (December 1978) 1802-1811.

55. McQuillan, J.M., Richer, I., and Rosen, E.C., The New Routing Algorithm for the ARPANET. *IEEE Transactions on Communications COM-28*, 5 (May 1980) 711-719.

56. Modarres, M., A Method of Predicting Availability Characteristics of Series-Parallel Systems. *IEEE Transactions on Reliability R-33*, 4 (October 1984) 309-312.

57. Nanya, T., and Kawamura, T., Error Secure/Propagating Concept and its Application to the Design of Strongly Fault-Secure Processors. *IEEE Transactions on Computers 37*, 1 (January 1988) 14-24.

58. Negrini, R., Sami, M., and Stefanelli, R., Fault Tolerance Techniques for Array Structures Used in Supercomputing. *Computer 19*, 2 (February 1986) 78-87.

59. Nikolaou, C.N., Reliability Issues in Distributed Systems. Technical Report RC9335, IBM San Jose Research Laboratory, San Jose, California, 1982.

60. Ousterhout, J.K., Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa. Technical Report CMU-CS-80-112, Carnegie-Mellon University, 1980.

61. Ozaki, B.M., Fernandez, E.B., and Gudes, E., Software Fault Tolerance in Architectures with Hierarchical Protection Levels. *IEEE Micro 8*, 4 (August 1988) 30-43.

62. Patki, V.B., Patki, A.B., and Chatterji, B.N., Reliability and Maintainability Considerations in Computer Performance Evaluation. *IEEE Transactions on Reliability R-32*, 5 (December 1983) 433-436.

63. Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Theil, G., LOCUS – A Network Transparent, High Reliability Distributed System. *Proceedings of the 8th Symposium on Operating System Principles*, Monterey, California, (15-17 December 1981) 169-177.

64. Randell, B., System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975) 220-232.

65. Raytheon Company, Advanced Onboard Signal Processor (AOSP) Phase IIB Development. Technical Report under Contract Number F30602-84-C-0094, Rome Air Development Center, Griffiss AFB, N.Y., 1987.

66. Raytheon Company, Interface Control Document for AOSP Brassboard System Applications. Technical Report under Contract Number F30602-84-C-0094, Rome Air Development Center, Griffiss AFB, N.Y., 1985

67. Raytheon Company, Computer Program Product Specification for Global Operating System. Technical Report G166873A, Raytheon Company, Sudbury, Massachusetts, 1 August 1983.

68. Rennels D.A., Fault Tolerant Computing: Issues, Examples, and Methodology. Technical Report CSD-870024, University of California Los Angeles, 1987.

69. Russell, C.H., and Waterman, P.J., Variations on Unix for Parallel-Processing Computers. *Communications of the ACM 30*, 12 (December 1987) 1048-1055.

70. Samson, J.R., Horrigan, F.A., Jagodnik, A.J., Soli, R.H., The Advanced Onboard Signal Processor (AOSP) - A Validated Concept. *Proceedings of the 9th DARPA Strategic Space Symposium*, Monterrey, California, (4-7 October 1983) 217-223.

71. Sarrazin, D.B., and Malek, M., Fault-Tolerant Semiconductor Memories. *Computer 17*, 8 (August 1984) 49-56.

72. Schlichting, R., Cristian, F., and Purdin, T., Mechanisms for Failure Handling in Distributed Programming Languages. Technical Report RJ5173, IBM San Jose Research Laboratory, San Jose, California, 1978.

73. Serlin, O., Fault-Tolerant Systems in Commercial Applications. *Computer 17*, 8 (August 1984) 19-30.

74. Shamir, E., and Upfal, E., A Probabilistic Approach to the Load Sharing Problem in Distributed Systems. Technical Report RJ5315, IBM San Jose Research Laboratory, San Jose, California, 1986.

75. Shooman, M.L., Software Reliability: A Historical Perspective. *IEEE Transactions on Reliability R-33*, 1 (April 1984) 48-55.

76. Shrivastava, S.K., Robust Distributed Programs. In Anderson, T. (ed.) *Resilient Computing Systems Volume I.* John Wiley & Sons, New York, New York, 1985. 102-121.

77. Siewiorek, D.P., Architecture of Fault-Tolerant Computers. *Computer 17*, 8 (August 1984) 9-18.

78. Siewiorek, D. P., and Swarz, R. S., *The Theory and Practice of Reliable System Design*. Digital Press, Bedford, Mass., 1982.

79. Sindhu, P.S., Distribution and Reliability in a Multiprocessor Operating System. Technical Report CMU-CS-84-125, Carnegie-Mellon University, 1984.

80. Spector, A.Z., Communication Support in Operating Systems for Distributed Transactions. Technical Report CMU-CS-86-165, Carnegie-Mellon University, 1986.

81. Srikanth, T.K., and Toueg, S., Optimal Clock Synchronization. *Journal of the ACM 34*, 3 (July 1987) 626-645.

82. Stankovic, J.A., A Perspective on Distributed Computer Systems. *IEEE Transactions on Computers C-33*, 12 (December 1984) 1102-1115.

83. Stankovic, J.A., Ramamritham, K., and Kohler, W.H., A Review of Current Research and Critical Issues in Distributed System Software. In Bhargava, B. (ed.), *Concurrency Control and Reliability in Distributed Systems*. Van Nostrand Reinhold Co., New York, New York, 1987. 14-47.

84. Svobodova, L., Client/Server Model of Distributed Processing. Technical Report RZ1350, IBM San Jose Research Laboratory, San Jose, California, 1985.

85. Tevanian, A., Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach. Technical Report CMU-CS-88-106, Carnegie-Mellon University, 1987.

86. Tevanian, A., and Smith, B., MACH: The Model for Future UNIX. *BYTE 14*, 11 (November 1989) 411-416.

87. Toy, W.N., Dual Versus Triplication Reliability Estimations. *AT&T Technical Journal 66*, 6 (November 1987) 15-20.

88. Turner, J., Reliability Analysis of a Planar-2 System with Relocatable Processes. CS685 Report, Texas A&M University Computer Science Department, 1989.

89. Varma, A., and Chalasani, S., Fault-Tolerance Analysis of One-Sided Crosspoint Switching Networks. Technical Report RC14240, IBM Research Division, Yorktown Heights, New York, 1988.

90. Von Neumann, J., Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. In Shannon, C.E., and McCarthy, J. (eds.), *Automata Studies*, Princeton University Press (1956) 43-98.

91. Walker, B.J., and Popek, G.J., A Transparent Environment. *BYTE 14*, 7 (July 1989) 225-233.

92. Wallace, J.J, and Barnes, W.W., Designing for Ultrahigh Availability: The Unix RTR Operating System. *Computer 17*, 8 (August 1984) 31-39.

93. Wensley J.H., et al., SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE 66*, 10 (October 1978) 1240-1255.

94. Williams, R.D., Johnson, B.W., and Roberts, T.E., An Operating System for a Fault-Tolerant Multiprocessor Controller. *IEEE Micro 8*, 4 (August 1988) 18-29.

95. Yau, S.S., An Approach to Distributed Computing System Software Design. *IEEE Transactions on Software Engineering SE-17*, 4 (July 1981) 427-436.

96. Zorpette, Z., Computers that are 'Never' Down. *IEEE SPECTRUM 22*, 4 (April 1985) 46-54.

APENDIX A

FTM SYSTEM PROTOCOLS

Appendix A details the protocols underlying the FTM system. These are catagorized as protocols for IC-to-FTM and FTM-to-FTM and FTM-to-IC Communications.

All protocols within the FTM system, whether IC-to-FTM, FTM-to-FTM, or FTM-to-IC, are messages which are formatted according to the following structure defined in the C language:

```
struct msgbuf {
   char  cmd[7];          /* message type                */
   char  from;            /* message from FTM or IC ?    */
   char  status[6];       /* action taken on msg so far */
   char  name_to[21];     /* target of message           */
   char  name_from[21];   /* originator of message       */
   char  seqnum[12];      /* message sequence number     */
   char  errno[12];       /* error code for exceptions   */
   char  mlen[12];        /* length of the next field    */
   char  msg[2049];       /* data portion of the msg     */
}
```

Before attempting to read the detailed descriptions of the individual protocol types, the following items of information should be read and understood:

1. The cmd field defines the protocol name. Unfortunately, the same protocol name in an IC-to-FTM message, an FTM-to-FTM message, and an FTM-to-IC message, allows for differing interpretations of the other fields. That is, the class of message (IC-to-FTM, FTM-TO-FTM, or FTM-to-IC) and protocol name together define the protocol type.

2. The from field is needed because an FTM component which has stored an arriving message will need to know the protocol type when the message gets processed, and as described in #1 above, the cmd field alone does not uniquely identify a protocol type.

3. The status field is used to indicate the further action needed in response to the receipt of an instance of this protocol type. A -1 indicates that the message has not yet been processed and a 0 indicates that the message has been fully processed. Originally, the protocols included some (status == 0) "no action" types (these simply checked for open logical connections) but the current protocol set includes only "action required" messages (status == -1). The FTM system components for some protocol types reset this field in the message (stored after receipt) to a positive integer value to keep track of partial completion of the actions required. While this is implementation (rather than protocol) related, these are described here to allow this section to be used in debugging the FTM system should a dump of the message queues be required.

4. Sometimes other fields are also modified during message storage by the receiving FTM system component. Again, this is implementation rather than protocol related, but is mentioned in the following descriptions (under the heading STORAGE:) for convenience.

5. The following individual protocol types are described in "close to" alphabetical order, within protocol class (first IC-to-FTM, then FTM-to-FTM, and finally FTM-to-IC). This forces a fair amount of duplication, but appears to be less confusing than the alternative orderings.

<u>PROTOCOL CLASS:</u>
    IC-to-FTM
cmd:
    ADV
from:
    I
status:
    -1 until acted upon.
    +k while k future responses from other FTMs remain outstanding.
     0 when all responses (ADVOK and/or ADVNO) have been received.
name_to:
    not used

name_from:
the advertised name by which the advertising application wishes to be identified.
seqnum:
the next sequential message sequence number (ordinarily will be 1).
errno:
not used
mlen:
the length of the next field.
msg:
the process ID of the application, which may be used by the FTM to communicate with the application through other than the normal FTM-to-IC message channel. For example, the FTM may send interrupts (or signals).
DISCUSSION:
An application has requested connection to the FTM system by attempting to advertise a name. As a result, the Intercept Code has connected as a client to the local FTM, and the first message sent by the IC to the local FTM defines the name by which the application wishes to be known throughout the FTM system. The FTM system must now assure that the requested name will be unique, so the local FTM will contact all other FTMs (with an ADV message of the FTM-to-FTM class), and will maintain status > 0 until all other FTMs have responded.
STORAGE:
The msg field is extended with two values: one for the logical connection identifier between the local ftm and the application, and the other with information about the backup system selected for the application.

PROTOCOL CLASS:
IC-to-FTM
cmd:
CLOSE
from:
I
status:
-1
name_to:
not used
name_from:
the name by which the application is currently identified.
seqnum:
the next sequential message sequence number.
errno:
not used
mlen:
0
msg:
not used
DISCUSSION:
An application wishes to disassociate itself from the FTM system. The local FTM will propagate the information to all other FTMs through FTM-TO-FTM KILLN messages.
STORAGE:
no changes

PROTOCOL CLASS:
IC-to-FTM
cmd:
STATUS
from:
I
status:
-1

name_to:
not used
name_from:
the advertised name of the requesting application.
seqnum:
the next sequential message sequence number.
errno:
not used
mlen:
0
msg:
not used
DISCUSSION:
The application is requesting status about the current FTM system. The exact status available will be implementation dependent, but should include such items as the physical systems in the current FTM configuration, links, workloads, other available resources, etc.
STORAGE:
no changes

PROTOCOL CLASS:
IC-to-FTM
cmd:
TO
from:
I
status:
-1 until acted upon.
+1 when the message has been forwarded to the next hop.
0 when response from the receiver has been received.
name_to:
the advertised name of the desired user message recipient (target).
name_from:
the advertised name of the user message sender (originator).
seqnum:
the next sequential message sequence number.
errno:
not used
mlen:
the length of the next field.
msg:
the user message. This need not be ascii, since it will be transmitted as a byte stream of known length.
DISCUSSION:
An application has requested (through a call to its Intercept Code interface) that a user message be delivered to another application. The IC embeds the message into this protocol type and sends it to the local FTM. The local FTM will forward the user message to the next hop in the path, either to a remote FTM or directly to the target application if it exists on the local system. A "hop" is the transmission of a message from a node to the next node in the path toward the target node.
STORAGE:
The name_to and name_from fields are swapped, to match the order in the eventual response message. The stored user message is unaltered so it can be checked against the future response from the target Intercept Code.

PROTOCOL CLASS:
IC-to-FTM
cmd:
TOOK
from:
I
status:
-1
name_to:
the advertised name of the user message sender (originator).
name_from:
the advertised name of the user message recipient (target).
seqnum:
a copy of seqnum in the IC-to-FTM TO message that this message is in response to.
errno:
not used
mlen:
the length of the next field
msg:
the user message. This is echoed back so it can be compared to the original message at each return hop.
DISCUSSION:
The message whose delivery has been requested by application A has been reliably delivered to the Intercept Code of target application B on a remote system. The IC in the target application sends this IC-to-FTM TOOK message back along the reverse path to so inform the FTM on the target system.
STORAGE:
not used

PROTOCOL CLASS:
IC-to-FTM
cmd:
TONO
from:
F
status:
-1
name_to:
the advertised name of the user message sender (originator).
name_from:
the advertised name of the desired user message recipient (target).
seqnum:
a copy of seqnum in the IC-to-FTM TO message that this message is in response to.
errno:
a code to indicate the reason why the message could not be delivered.
mlen:
the length of the next field.
msg:
the user message.
DISCUSSION:
This is the negative equivalent of the IC-to-FTM TOOK protocol type described above. For some reason (identified in the errno field) the message which application A requested be delivered to application B could not be reliably delivered.
STORAGE:
no changes

PROTOCOL CLASS:
IC-to-FTM
cmd:
WHERE
from:
I
status:
-1
name_to:
not used
name_from:
the advertised name of the requesting aplication.
seqnum:
the next sequential message sequence number.
errno:
not used
mlen:
the length of the next field.
msg:
the advertised name of the application whose location is being requested.
DISCUSSION:
An application is requesting the current physical location of another application. The FTM will respond with information from its database.
STORAGE:
no changes

PROTOCOL CLASS:
FTM-to-FTM
cmd:
ADV
from:
F
status:
-1 until acted upon.
+1 while awaiting confirmation that name_from is system-wide unique.
0 when confirmation (positive or negative) has arrived.
name_to:
not used
name_from:
the advertised name by which the advertising application wishes to be known.
seqnum:
the next sequential message sequence number.
errno:
not used
mlen:
the length of the next field.
msg:
three fields: the process ID of the advertising application, an identifier for the logical connection between the local FTM and the application, and information about the backup system selected for the application.
DISCUSSION:
Once a local FTM has received an IC-to-FTM ADV message from an application, it sends one of these FTM-to-FTM ADV messages to each other FTM in the current FTM system. Each of these other FTMs will eventually respond with an FTM-to-FTM ADVOK or ADVNO message, depending on whether the name is unique to it.
STORAGE:
The msg field is extended with one value: an identifier for the logical connection between the receiving FTM and the sending FTM.

PROTOCOL CLASS:
FTM-to-FTM
cmd:
ADVOK
from:
F
status:
-1
name_to:
the name whose advertisement this message is in response to.
name_from:
not used
seqnum:
the next sequential message sequence number.
errno:
not used
mlen:
the length of the next field.
msg:
three fields: the process ID of the advertising application, an identifier for the logical connection between the local FTM and the application, and information about the backup system selected for the application.
DISCUSSION:
This message implements phase 1 of the 2-phase commit protocol required to declare a name as unique within the FTM system. An FTM has received a request to advertise a name and has broadcast the FTM-to-FTM ADV message to all other FTMs. Each broadcast recipient who agrees that the name is unique replies with an FTM-to-FTM ADVOK message. The second phase of the 2-phase commit will occur only if all broadcast recipients eventually agree.
STORAGE:
no changes

PROTOCOL CLASS:
FTM-to-FTM
cmd:
ADVNO
from:
F
status:
-1
name_to:
the name whose advertisement this message is in response to.
name_from:
not used
seqnum:
the next sequential message sequence number.
errno:
indicates that the name is not unique, or identifies a miscellaneous error.
mlen:
the length of the next field.
msg:
three fields: the process ID of the advertising application, an identifier for the logical connection between the local FTM and the application, and information about the backup system selected for the application.
DISCUSSION:
This message is the negative equivalent of the FTM-to-FTM ADVOK message described above. An FTM has received a request to advertise a name and has broadcast the FTM-to-FTM ADV message to all other FTMs. This broadcast recipient replies with an FTM-to-FTM ADVNO message to reject the request.
STORAGE:
no changes

PROTOCOL CLASS:
FTM-to-FTM
cmd:
ADVOKY
from:
F
status:
-1
name_to:
the name whose advertisement this message is in response to.
name_from:
not used
seqnum:
the next sequential message sequence number.
errno:
not used
mlen:
0
msg:
not used
DISCUSSION:
This message implements phase 2 of the 2-phase commit protocol required to declare a name as unique within the FTM system. An FTM has received a request to advertise a name and has broadcast the FTM-to-FTM ADV message to all other FTMs. Each broadcast recipient has agreed that the name is unique and has thereby replied with an FTM-to-FTM ADVOK message. Since the original FTM has received all ADVOK replies to its FTM-to-FTM ADV broadcast, the name is indeed unique throughout the FTM system, so this FTM-to-FTM ADVOKY message is broadcast so that all FTMs will treat the name as a unique identifier.
STORAGE:
not used

PROTOCOL CLASS:
FTM-to-FTM
cmd:
ADVOKN
from:
F
status:
-1
name_to:
the name whose failed advertisement this message is in response to.
name_from:
not used
seqnum:
the next sequential message sequence number.
errno:
indicates that the name is not unique, or identifies a miscellaneous error.
mlen:
0
msg:
not used
DISCUSSION:
This is the negative equivalent of the FTM-to-FTM ADVOKY protocol type described above. Phase 2 of the 2-phase commit protocol has failed: the original FTM has broadcast an advertised name to all other FTMs, but at least one of these FTMs has replied with an FTM-to-FTM ADVNO and errno code to indicate duplicate name. This can happen, for example, if some other application somewhere in the distributed FTM system is simultaneously attempting to advertise an identical name. So, the original FTM uses this protocol type (FTM-to-FTM ADVOKN) to inform all other FTMs that it is retracting its partially completed attempt to advertise the name.

STORAGE:
not used

PROTOCOL CLASS:
FTM-to-FTM
cmd:
KILLN
from:
I if the disconnect was detected locally
F if the disconnect was detected by a remote FTM
status:
-1
name_to:
not used
name_from:
the previously advertised name which should now be deleted.
seqnum:
the next sequential message sequence number.
errno:
0 if this is the result of an ftmclose call.
-1 if this is the result of a disconnected application socket.
-2 if this is the result of a remote FTM failure.
mlen:
0
msg:
not used
DISCUSSION:
The sending FTM is requesting that a name which has been successfully advertised as unique in the past should now be completely deleted from all FTMs in the FTM system. This could be because the application has requested disconnection from the FTM system, the application has failed and the name must be deleted prior to re-advertisement at its backup node, or a complete node (or at least its FTM) has failed and the FTM which detected the failure is requesting deletion of all names advertised by the failing FTM.
STORAGE:
not used

PROTOCOL CLASS:
FTM-to-FTM
cmd:
TO
from:
F
status:
-1 until acted upon.
+1 when the message has been forwarded to the target IC.
0 when response from the target has been received.
name_to:
the advertised name of the desired user message recipient (target).
name_from:
the advertised name of the user message sender (originator).
seqnum:
a copy of seqnum in the IC-to-FTM TO message that this message is in response to.
errno:
not used
mlen:
the length of the next field.
msg:
the user message. This need not be ascii, since it will be transmitted as a byte stream of known length.

DISCUSSION:

An application A has requested that a message be delivered to an application B. The FTM local to application A received the request (through an IC-to-FTM TO message) and determined that application B is on a different system. So, it sends this FTM-to-FTM TO message to forward the application's message to the FTM at the target system.

STORAGE:

The name_to and name_from fields are swapped, to match the order in the eventual response message.

PROTOCOL CLASS:

FTM-to-FTM

cmd:

TOOK

from:

F

status:

-1

name_to:

the advertised name of the user message sender (originator).

name_from:

the advertised name of the user message recipient (target).

seqnum:

a copy of seqnum in the IC-to-FTM TO message that this message is in response to.

errno:

not used

mlen:

the length of the next field

msg:

the user message. This is compared against the FTM-to-FTM TO message which has been retained (with status == +1) to assure that the message has been delivered to the target and echoed back correctly.

DISCUSSION:

The message whose delivery has been requested by application A has been reliably delivered to the Intercept Code of target application B on a remote system. The FTM on the target system uses this FTM-to-FTM TOOK message to so inform the FTM on the original sending system.

STORAGE:

not used

PROTOCOL CLASS:

FTM-to-FTM

cmd:

TONO

from:

F

status:

-1

name_to:

the advertised name of the user message sender (originator).

name_from:

the advertised name of the desired user message recipient (target).

seqnum:

a copy of seqnum in the IC-to-FTM TO message that this message is in response to.

errno:

a code to indicate the reason why the message could not be delivered.

mlen:

the length of the next field.

msg:

the user message.

DISCUSSION:
This is the negative equivalent of the FTM-to-FTM TOOK protocol type described above. For some reason (identified in the errno field) the message which application A requested be delivered to application B could not be reliably delivered.
STORAGE:
no changes

PROTOCOL CLASS:
FTM-to-IC
cmd:
ADVOK
from:
F
status:
-1
name_to:
the name whose advertisement this message is in response to.
name_from:
not used
seqnum:
the next sequential message sequence number.
errno:
not used
mlen:
0
msg:
not used
DISCUSSION:
The advertise operation has completed successfully. This message from the local FTM informs the Intercept Code of the requesting application that the requested name has been found to be unique and has been successfully advertised throughout the FTM system. The application may now request any available FTM system services.
STORAGE:
The Intercept Code has blocked while awaiting response to an ADV message. This return message is handled immediately without ever being stored.

PROTOCOL CLASS:
FTM-to-IC
cmd:
ADVNO
from:
F
status:
-1
name_to:
the name whose advertisement this message is in response to.
name_from:
not used
seqnum:
the next sequential message sequence number.
errno:
indicates that the name is not unique, or identifies a miscellaneous error.
mlen:
0
msg:
not used
DISCUSSION:
This message is the negative equivalent of the FTM-to-IC ADVOK message described above. The advertise operation has not completed successfully. This message from the local FTM informs the Intercept Code of the requesting application that the

operation has failed and that the application has not been connected to the FTM system. The reason for this failure (duplicate name or miscellaneous system error) is indicated by the errno field.

STORAGE:

The Intercept Code has blocked while awaiting response to an ADV message. This return message is handled immediately without ever being stored.

PROTOCOL CLASS:

FTM-to-IC

cmd:

CLOSOK

from:

F

status:

-1

name_to:

the advertised name of the application which has just been disconnected.

name_from:

not used.

seqnum:

the next sequential message sequence number.

errno:

not used

mlen:

0

msg:

not used

DISCUSSION:

The application has requested that it be disconnected from the FTM system. Its Intercept Code has forwarded this request to the local FTM, which has performed the operation locally (and remotely through FTM-to-FTM KILLN messages). Since the IC has preprocessed the original request, including consistency checking, this request will always succeed.

STORAGE:

The Intercept Code has blocked while awaiting response to a CLOSE message. This return message is handled immediately without ever being stored.

PROTOCOL CLASS:

FTM-to-IC

cmd:

STATOK

from:

F

status:

-1

name_to:

the advertised name of the application which has just been disconnected.

name_from:

seqnum:

the next sequential message sequence number.

errno:

not used

mlen:

the length of the next field.

msg:

the status information. The length and contents will be implementation dependent, but should include such information as current nodes and links in the FTM system, miscellaneous available system resources, current workloads, etc.

DISCUSSION:

The application has requested that it be given current status information by the

FTM system. Its Intercept Code has forwarded this request to the local FTM, which has performed the operation locally. Since the IC has preprocessed the original request, including consistency checking, this request will always succeed.

STORAGE:

The Intercept Code has blocked while awaiting response to a STATUS message. This return message is handled immediately without ever being stored.

PROTOCOL CLASS:

FTM-to-IC

cmd:

TO

from:

F

status:

-1

name_to:

the advertised name of the desired user message recipient (target).

name_from:

the advertised name of the user message sender (originator).

seqnum:

a copy of seqnum in the IC-to-FTM TO message that this message is in response to.

errno:

not used

mlen:

the length of the next field.

msg:

the user message. This need not be ascii, since it will be transmitted as a byte stream of known length.

DISCUSSION:

An application has requested (through a call to its Intercept Code interface) that a user message be delivered to another application. That message has been forwarded through the FTM system and is now being delivered to the IC at the target application.

STORAGE:

The entire message is stored unchanged (on a queue) by the target IC. The target IC will immediately respond (with an IC-to-FTM TOOK or TONO) since the originator IC is blocking awaiting confirmation that the user message has been delivered to the target IC. The application may read the message anytime in the future.

PROTOCOL CLASS:

FTM-to-IC

cmd:

TOOK

from:

F

status:

-1

name_to:

the advertised name of the user message sender (originator).

name_from:

the advertised name of the user message recipient (target).

seqnum:

a copy of seqnum in the IC-to-FTM TO message that this message is in response to.

errno:

not used

mlen:

the length of the next field.

msg:

the user message. This is echoed back so it can be compared to the original message at each return hop.

DISCUSSION:

The message whose delivery has been requested by application A has been reliably delivered to the Intercept Code of target application B on a remote system. The IC in the target application has echoed back with an IC-to-FTM TOOK message, which has been forwarded appropriately and is now being sent along the final return path hop to the originator application.

STORAGE:

The Intercept Code has blocked while awaiting response to a TO message. This return message is handled immediately without ever being stored.

PROTOCOL CLASS:

FTM-to-IC

cmd:

TONO

from:

F

status:

-1

name_to:

the advertised name of the user message sender (originator).

name_from:

the advertised name of the user message recipient (target).

seqnum:

a copy of seqnum in the IC-to-FTM TO message that this message is in response to.

errno:

a code to indicate the reason why the message could not be delivered.

mlen:

the length of the next field.

msg:

the user message. This is echoed back so it can be compared to the original message at each return hop.

DISCUSSION:

This is the negative equivalent of the FTM-to-IC TOOK protocol type described above. For some reason (identified in the errno field) the message which application A requested be delivered to application B could not be reliably delivered.

STORAGE:

The Intercept Code has blocked while awaiting response to a TO message. This return message is handled immediately without ever being stored.

PROTOCOL CLASS:

FTM-to-IC

cmd:

WHEROK

from:

F

status:

-1

name_to:

the advertised name of the application which has just been disconnected.

name_from:

seqnum:

the next sequential message sequence number.

errno:

not used

mlen:

the length of the next field.

msg:

the current physical system on which the application in question resides.

DISCUSSION:

The application has requested that it be given the current physical system on

which a certain application, identified by its advertised name, resides. Its Intercept Code has forwarded this request to the local FTM, which has performed the operation locally. Since the IC has preprocessed the original request, including consistency checking, this request will always succeed.

STORAGE:

The Intercept Code has blocked while awaiting response to a WHERE message. This return message is handled immediately without ever being stored.

APENDIX B

FTM PROTOTYPE DISTRIBUTION LIBRARY

The FTM system is available for distribution on a diskette. This diskette contains three files: *INSTRUC*, *FTM.LIS*, and *FTM.TAR*. *INSTRUC* is a text file which describes how *FTM.TAR* was created, so that it can be restored to a UNIX system. *FTM.LIS* is a list of the individual files bundled into *FTM.TAR*. The restored *FTM.TAR* will consist of the files as listed in *FTM.LIS*. These files are printed in Appendix B and Appendix C. The files in Appendix B can be used to create a customized FTM system, while the files in Appendix C result in validation and performance tests – the same tests used to generate the statistics of Chapter V.

Once the distribution library has been restored to a UNIX system, the steps necessary to create a customized FTM system are as follows:

1. Edit file *header.h* to update three #define constants: First, PORT must be set to a UNIX communication port which is available on all local UNIX systems which may become part of a running FTM distributed system. For convenience, it may be desireable to *register* this port to the FTM system so that it is unavailable to other users. Second, FTMPATH must be set to contain the complete path to the directory into which the FTM distribution library has been restored. Third, SOCKPATH must be set to contain the complete path to a directory into which the FTM system can create temporary files. This directory may be replicated on each individual node or may be a single directory available to each node within the FTM system.

2. Invoke standard UNIX utility "make" to create the required FTM system executables and libraries.
3. Edit the file *ftms.lis* to reflect the local configuration for the distributed FTM system and file *ftmbkup.lis* to reflect the user programs for which automatic restart/relocation is desired. This information must be available throughout the distributed FTM system. Thus, if the directory into which the FTM distribution library has been restored is not file served throughout the various nodes, then *ftms.lis* and *ftmbkup.lis* must be replicated on each node. The FTM executable (file *ftm*, created during step 2) similarly must be available at each node.
4. That's it - you should now have an operational FTM system ready to run. Please read the file *Readme* (the first file printed in this appendix) which contains complete startup and user interface details.

```
/*****************************************************************************/
/*                                                                           */
/*   Readme                                                                  */
/*                                                                           */
/*   Purpose - application programmers instruction guide                     */
/*                                                                           */
/*****************************************************************************/
```

FTM and INTERCEPT LIBRARY V03.04

CHANGES FROM V03.03:

(1) - the ftms no longer crash together. If one goes down, the others continue to interact. Any connected applications on the failing ftm with a defined backup system are restarted on that backup system.

***** THE FOLLOWING CHANGE PRODUCES AN INCOMPATABILITY *****
(2) - the bkup.lis file is no longer used. It has been replaced by the ftmbkup.lis file.

***** THE FOLLOWING CHANGE PRODUCES AN INCOMPATABILITY *****
(3) - the interface call to ftmwhere has been changed to add another parameter.

DIRECTORY: /u/auvusers1/auvschi/ftm/

CONTENTS:

Readme	- This file.
Makefile	- The file used as input to UNIX program "make" to create the libraries and executables in this distribution library.
Lint	- This file can be used as input to UNIX program "make" in order to perform consistency checking beyond that of the C compiler.
header.h	- Global definitions and list of other header files used in the FTM and Intercept Code routines. In general, one can update the FTM system (to change parameter values and/or define paths to rehost the FTM system) by updating the well-commented entries in this file and then invoking the standard UNIX "make" program.
ftms.lis	- A sample text file containing the list of active systems for the next FTM system startup.
ftmbkup.lis	- A sample text file containing information the ftm can use to restart programs on their backup systems.
rkill	- A utility shell script used locally to send signals to remote processes. This is used by FTM shutdown program: ftmkill. rkill would have to be modified to run on a different local area network with different node names.
ztest*.c	- 4 sample programs that communicate with each other. These can be used to verify that a generated FTM system performs basic operations.
z*.c	- Sample programs to generate FTM system performance statistics.
*.c	- The source code for the FTM system.

THE FOLLOWING ARE NOT INCLUDED IN THIS DISTRIBUTION LIBRARY, BUT ARE GENERATED BY UNIX PROGRAM "make":

intlib.a	- The intercept library, to be linked into programs.
ftm	- The fault tolerant monitor executable.

```
ftmstart    - Command to automatically start the ftms.
rundaemon   - Called by ftmstart to start remote ftm programs as
              UNIX daemons.
ftmkill     - Calls shell script rkill (described above) to shut
              down remote ftm programs.
ALSO GENERATED BY PROGRAM "make" ARE OTHER LIBRARIES AND OBJECT
FILES USED ONLY BY PROGRAM "make" AND RETAINED ONLY TO SPEED UP
FUTURE FTM SYSTEM GENERATIONS. ALSO , *.BAK FILES ARE GENERATED
BY PROGRAM "indent" WITHIN "make". THESE CAN GENERALLY BE
DELETED AFTER "make" COMPLETES SUCCESSFULLY.
```

INTRODUCTION:

The purpose of the ftm and intercept code is to allow applications to send messages to each other without the need to know which physical machine is currently running any particular application. Applications may be terminated (either normally or killed) and restarted on the same or another machine.

PROCEDURE:

In order to use this routing, the following steps must be followed:

(1) - Link intlib.a into your C or LISP program.
(2) - Edit ftms.lis to include the systems on which you will be running your tasks. It is not mandatory that a user task must actually be run on each of these systems.
(3) - Run "ftm" on each of the systems included in the ftms.lis file. It IS MANDATORY that ftm be run on each of these systems. They should all be started within 60 seconds (a #define constant set in file: header.h) of each other. They may be run in the background - the only output from each ftm is the line (to syserr):

```
FTM: xxxxxx FULLY CONNECTED - START APPLICATIONS
```

which will appear only after an ftm has connected to all the other ftms. There is no other output except error messages to syserr.
(3a)- You may run "ftmstart" to automatically start the ftms. This command will read the ftms.lis file to determine which systems on which to start ftms. A daemon is invoked to start each of these ftms, and the ftm messages are saved in a file (defined in file: header.h).
(4) - Edit ftmbkup.lis to reflect any programs you want to be automatically restarted on their backup systems.
(5) - Start the applications. At any time, an application may be terminated and restarted on the same or any connected system.
(6) - an application must call adv(name) as its first call to the intercept routines. See INTERFACE below for details of calls.
(7) - When you are done, kill the ftms. They never stop on their own, unless an ftm detects an error that it cannot handle.
(7a)- You may run "ftmkill" to kill all the ftms in this ftm system.

GLOBAL NAMES:

The following global names are used for procedures (P) and variables (V) within the intercept code. They should NOT be used as global names by any routines with which the intercept routines will be linked.

```
P - findbuf
V - ftmsock
V - ftmtime
V - ftm_ic
P - getmsgs
```

```
 V - intadvyet
 V - intmyname
 V - intpidme
 V - int_msgs
 V - intseqnum
 V - msgfirst
 P - msgfrmlist
 V - msglast
 P - msgtolist
 P - namlookup
 P - recv_msg
 P - recv_pkt
 P - send_msg
*P - setcap
*P - strlcpy
*P - strnlen
```

Actually, the procedures above marked *P are utility routines which may be called directly if desired. The definitions are:

```
void setcap (buf, len)        /*  The buffer buf of length len has all   */
char        *buf;             /*  lower case characters converted to     */
int          len;             /*  upper case.                            */

char *strlcpy (s1, s2, len)   /*  This call operates as strncpy()        */
char          *s1,            /*  is described in the blue C book        */
              *s2;            /*  by Kochan.  The standard SUN UNIX      */
int            len;           /*  strncpy() routine is different.        */

int strnlen (s1, len)         /*  This routine returns:                  */
char        *s1;              /*     min (strlen (s1), len)              */
int          len;             /*                                         */
```

INTERFACE:

```
int     adv (name)
char        *name;
```

At present, name can be any ascii string up to 20 characters. If it is of length 1, then it cannot be "*". If name is longer than 20 characters, then only the first 20 are used. Internally, the capitalized version of name is maintained.

```
rc ==  0: successful return, the name has been advertised to all ftms.

rc == -1: errno == EWOULDBLOCK => name is currently known in the system; the
                                  system will not wait for the other application
                                  using this name to terminate.
          errno == EALREADY    => this process has already successfully
                                  advertised a name.  Only one name per process
                                  is allowed.
          errno == EINVAL      => the name parameter is zero length or is "*"
          errno == ENOTCONN    => the connection to the local ftm has been lost;
                                  this is an uncorrectable error.
          errno == EFAULT      => misc. uncorrectable error.
```

```
int     copycf (name, filename)
char           *name,
               *filename;
```

name is as described above. filename is the full path to a local file. This file is remote copied to the system wich is the current backup system for the application which has advertised name. Ordinarily, name is the advertised name for the calling application, so that copycf serves to provide a remote copy of a local critical file at the backup system.

```
rc ==  0: successful return, the file has been copied.

rc == -1: errno == EWOULDBLOCK => name is currently unknown in the system; the
                                  system will not wait for some application to
                                  advertise this name.
          errno == EINVAL      => a parameter is zero length or too long
          errno == ENOTCONN    => the connection to the local ftm has been lost;
                                  this is an uncorrectable error.
          errno == any other   => errno is the error code from the system
                                  call to invoke a rcp routine. Possible
                                  errors include non-existent local file
                                  or unreachable target directory.
```

```
int  from (name, buf, len)
char      *name,
          *buf;
int        len;
```

name is as described above. buf points to the message buffer into which the message will be placed. len is the size of the buffer. If the message is too long to fit, the remainder will be discarded. The message placed into buf will be the oldest current message from the application which advertised name. name is case-insensitive.

Option: if name points to the single character ascii STRING "*", then the message placed into buf will be the oldest current message from any application. In this case name becomes a value-return parameter. It receives the advertised name of the message sender. Note that the capitalized version of the advertised name is returned. This may differ from the actual originally advertised name.

```
rc >=  0: successful return, rc bytes were actually placed into buf.

rc == -1: errno == EWOULDBLOCK => no messages have been received from a sender
                                  who has advertised name.  It may or may not
                                  be that such a sender currently exists in any
                                  connected system.
          errno == EINVAL      => the name parameter is zero length or the len
                                  parameter is negative.
          errno == ENOTCONN    => there are no messages which fit the requested
                                  criteria in the message buffer.  In addition,
                                  the connection to the local ftm has been lost;
                                  this is an uncorrectable error.
          errno == EFAULT      => misc. uncorrectable error.
```

```
int  ftmclose ()
```

This call removes the advertised name from the ftm tables. If the program terminates without calling ftmclose() then the ftm considers it to be an abnormal termination, which means that if a backup system has been defined in the bkup.lis file then the ftm on that backup system will attempt to restart the program. If this program is not in the bkup.lis file, then it does not matter whether or not ftmclose() is called before the program terminates.

```
rc ==  0: successful return.

rc == -1: errno == ENOTCONN    => the connection to the local ftm has been lost
```

```
                                    during the ftmclose call.  This is an
                                    uncorrectable error.  Note that the connection
                                    will always be broken as a result of this
                                    ftmclose call.
          errno == EFAULT       => misc. uncorrectable error.
-----------------------------------------------------------------------------
int  ftmstatus (pnum, sunstat, mirror)
int             pnum,
               *sunstat,
               *mirror;

pnum is the number of following parameters.  This is for future enhancements
to this call in which additional status items will be returned.  All parameters
after pnum are return parameters.  This means:

        pnum <= 0 => nothing will be returned.
        pnum == 1 => sunstat will be returned.
        pnum == 2 => sunstat and mirror will be returned.

sunstat has bits set to indicate active systems with a connected ftm.
mirror indicates the status of the disk mirroring.

sunstat: BIT 0  => auvsun0
         BIT 1  => auvc1
         BIT 2  => auvc2
         BIT 3  => auvc3
         BIT 4  => auvc4
         BIT 5  => auvc5
         BIT 6  => auvc6
         BIT 7  => auvc7
         BIT 8  => auvc8
         BIT 9  => auvc9
         BIT 10 => auvc10
         BIT 11 => auvc11
         BIT 12 => auvc12
         BIT 13 => auvc13
         BIT 14 => auvc14
         BIT 15 => auvc15
         BIT 16 => auvc16
         BIT 17 => auvsun1

mirror == 0 => DRIVE 0 DOWN   /* MEANINGLESS UNTIL DISK MIRRORING INSTALLED */
       == 1 => DRIVE 1 DOWN
       == 2 => BOTH DRIVES UP

rc ==  0: successful return.

rc == -1: errno == ENOTCONN    => the connection to the local ftm has been lost;
                                  this is an uncorrectable error.
          errno == EFAULT      => misc. uncorrectable error.

NOTE - the above status information will probably be of little or no use if
       the FTM system has been generated on a different distributed system.
       In this case the status message handler (hndlstatus.c) should be
       modified to provide useful status information with respect to the
       current hardware configuration.
-----------------------------------------------------------------------------
int  ftmwhere (name, buf1, buf2)
char          *name,
              *buf1,
              *buf2;
```

name is as described above. buf1 points to a buffer into which will be placed the name of the system on which the process that has advertised name is currently running. If there is no such process, then buf1 receives a null terminated zero length string (ie: a single hex zero byte). Our current system names require a buffer of length 8. For example: auvsun1 requires 7 bytes for the characters and 1 byte for the null terminator. It should be noted that SUN UNIX allows for system names of up to 64 characters (MAXHOSTNAMELEN in header file: /usr/include/sys/param.h), which would require a buffer of length 65. buf2 is similar, except that it receives the name of the backup system chosen by the ftm for the process.

```
rc >=  0: successful return.

rc == -1: errno == EINVAL       => the name parameter is zero length or is "*"
          errno == ENOTCONN     => the connection to the local ftm has been lost;
                                   this is an uncorrectable error.
          errno == EFAULT       => misc. uncorrectable error
-------------------------------------------------------------------------------
int     to (name, buf, len)
char       *name,
           *buf;
int         len;
```

name is as described above. buf points to the message buffer to be sent. If len is non-negative, it is interpreted as the length of the buffer and EXACTLY that many bytes are sent (this may include NULLs or ANY other bytes). If len is negative, then buf is assumed to contain an ascii string, and that string (including the NULL terminator byte) is sent. A zero length string may be sent and will be handled correctly. name is case-insensitive. At present, the maximum message length supported is 2048 bytes - any excess will be truncated. Since only the specified length is physically transmitted, it is wasteful of system resources to pad your buffer. An application may send a message to itself.

```
rc >=  0: successful return, rc bytes were actually transmitted.

rc == -1: errno == EWOULDBLOCK => name is currently unknown in the system.  The
                                  system will not wait for some other
                                  application to advertise this name.
          errno == EINVAL       => the name parameter is zero length or is "*"
          errno == ENOTCONN     => the connection to the local ftm has been lost;
                                   this is an uncorrectable error.
          errno == EFAULT       => misc. uncorrectable error.
-------------------------------------------------------------------------------
```

RESTART:

Once a program has successfully called adv(name) it may be automatically restarted by the ftm. If the name is listed in the ftmbkup.lis file, along with the path name for the program, then upon abnormal program termination the program will be restarted on the backup system by the backup system's ftm. See the comments in the bkup.lis file for format details.
Note the following:

(1) - the program is restarted after abnormal termination. The ftm defines abnormal termination as follows: The program has successfully called adv(name) and has terminated without successfully calling ftmclose().

(2) - the ftmbkup.lis file may be edited at any time and the newly saved version is used for future program terminations.

(3) - the only error in the ftmbkup.lis file that the ftm will detect is if

the path name to the program is not an executable file.

(4) - no arguments will be passed to the restarted program.

(5) - the control terminal for the restarted program will be inherited from the ftm on the restart system. If the restarted program needs its own terminal session to operate then it must handle this itself. The restarted program also inherits lots of other stuff from the ftm (like process id, working directory, etc.) See the execve(2) system call for full details.

RESTRICTIONS:

(1) fork() - Because the ftm needs to know from which process each message is received and to which process each message is sent, two processes cannot share the same advertised name. The application program can try to get around this restriction by a call to fork() following a call to adv(name). This means that the ftm would now be connected to two different processes through the same advertised name. Since the ftm would have no way of knowing to which process a message should be directed, the intercept library copy in the forked child resets itself. The following should be noted:
   (1) - only the process that calls adv(name) should subsequently call any of the other intercept library interface routines.
   (2) - any child process should advertise its own unique name if it needs a name to communicate by.
   (3) - a child process may communicate with its parent provided each has advertised a unique name,.
   (4) - there is nothing wrong with calling fork() after adv(name) as long as the child does not expect to communicate using that particular name.

(2) signals - The intercept code uses SIGUSR1. Therefore, linked programs must not issue, block, or handle SIGUSR1 signals. Also, the intercept code ignores EPIPE signals. If linked programs must provide a signal handler for broken pipes, then this handler should NOT call intercept code routines if the broken pipe is not on a user-initiated socket connection.

KNOWN BUGS:

(1) The backup system for an application is not dynamic. So, if the backup system for an application goes down while the application is running then the application will continue running, but without a backup system.

```
#*****************************************************************************/
#                                                                             /
#   Makefile                                                                  /
#                                                                             /
#   Purpose - create the FTM and the Intercept Library with this file as input /
#             to standard UNIX utility "make".                                /
#                                                                             /
#*****************************************************************************/

FTMOBJS = conall.o contimout.o ifget.o ffget.o findbuf.o ftmisdown.o hndladvf.o
          hndladvi.o hndladvok.o hndladvoky.o hndlclose.o hndlkilln.o
          hndlstatus.o hndlto.o hndltook.o hndlwhere.o msgdump.o msgfrmlist.o
          msghndl.o msgtolist.o namfrmlist.o namlookup.o namtolist.o nline.o
          pick_bkup.o recv_msg.o recv_pkt.o send_msg.o setcap.o strlcpy.o

INTOBJS = adv.o copycf.o findbuf.o from.o ftmclose.o ftmstatus.o ftmwhere.o
          getmsgs.o msgfrmlist.o msgtolist.o recv_msg.o recv_pkt.o send_msg.o
          setcap.o strlcpy.o strnlen.o to.o

INDENTPARMS = -bacc -bc -ci3 -d99 -nfcl -i3 -l80 -lp -npro -npsl -sc -v

all:    ftm ftmkill ftmstart rundaemon ftmtests tests

ftm:    ftm.o ftmlib.a
        cc ftm.o ftmlib.a -o ftm

ftmkill:  ftmkill.o
        cc ftmkill.o -o ftmkill

ftmstart:  ftmstart.o nline.o
        cc ftmstart.o nline.o -o ftmstart

ftmtests:  zadvclose zdummy zflop zstatus ztest2 ztest4 ztestc ztestd zto zwhere

rundaemon:  rundaemon.o
        cc rundaemon.o -o rundaemon

tests:  zzdummy_in zzdummy_un zzto_in zzto_un

zadvclose:  zadvclose.o intlib.a /lib/libm.a
        cc zadvclose.o intlib.a -lm -o zadvclose

zdummy: zdummy.o intlib.a
        cc zdummy.o intlib.a -o zdummy

zflop:  zflop.o intlib.a /lib/libm.a
        cc zflop.o intlib.a -lm -o zflop

zstatus:  zstatus.o intlib.a /lib/libm.a
        cc zstatus.o intlib.a -lm -o zstatus

ztest2: ztest2.o intlib.a
        cc ztest2.o intlib.a -o ztest2

ztest4: ztest4.o intlib.a
        cc ztest4.o intlib.a -o ztest4

ztestc: ztestc.o intlib.a
        cc ztestc.o intlib.a -o ztestc

ztestd: ztestd.o intlib.a
        cc ztestd.o intlib.a -o ztestd
```

```
zto:    zto.o intlib.a /lib/libm.a
        cc zto.o intlib.a -lm -o zto

zzdummy_in:  zzdummy_in.o
        cc zzdummy_in.o -o zzdummy_in

zzdummy_un:  zzdummy_un.o
        cc zzdummy_un.o -o zzdummy_un

zzto_in:  zzto_in.o /lib/libm.a
        cc zzto_in.o -lm -o zzto_in

zzto_un:  zzto_un.o /lib/libm.a
        cc zzto_un.o -lm -o zzto_un

zwhere: zwhere.o intlib.a /lib/libm.a
        cc zwhere.o intlib.a -lm -o zwhere

ftmlib.a:  $(FTMOBJS)
        ar rcv ftmlib.a $(FTMOBJS)
        ranlib ftmlib.a

intlib.a:  $(INTOBJS)
        ar rcv intlib.a $(INTOBJS)
        ranlib intlib.a

header.h:  /usr/include/errno.h          /usr/include/fcntl.h
           /usr/include/malloc.h         /usr/include/math.h
           /usr/include/netdb.h          /usr/include/signal.h
           /usr/include/string.h         /usr/include/netinet/in.h
           /usr/include/sys/resource.h   /usr/include/sys/socket.h
           /usr/include/sys/time.h       /usr/include/sys/types.h
           /usr/include/sys/un.h         /usr/include/sys/wait.h
        touch header.h

adv.o:  adv.c header.h
        -indent ${INDENTPARMS} adv.c
        cc -c adv.c

conall.o:  conall.c header.h
        -indent ${INDENTPARMS} conall.c
        cc -c conall.c

contimout.o:  contimout.c header.h
        -indent ${INDENTPARMS} contimout.c
        cc -c contimout.c

copycf.o:  copycf.c header.h
        -indent ${INDENTPARMS} copycf.c
        cc -c copycf.c

ffget.o:  ffget.c header.h
        -indent ${INDENTPARMS} ffget.c
        cc -c ffget.c

findbuf.o:  findbuf.c header.h
        -indent ${INDENTPARMS} findbuf.c
        cc -c findbuf.c

from.o: from.c header.h
        -indent ${INDENTPARMS} from.c
```

```
        cc -c from.c

ftm.o:  ftm.c header.h
        -indent ${INDENTPARMS} ftm.c
        cc -c ftm.c

ftmclose.o:  ftmclose.c header.h
        -indent ${INDENTPARMS} ftmclose.c
        cc -c ftmclose.c

ftmisdown.o:  ftmisdown.c header.h
        -indent ${INDENTPARMS} ftmisdown.c
        cc -c ftmisdown.c

ftmkill.o:  ftmkill.c header.h
        -indent ${INDENTPARMS} ftmkill.c
        cc -c ftmkill.c

ftmstart.o:  ftmstart.c header.h
        -indent ${INDENTPARMS} ftmstart.c
        cc -c ftmstart.c

ftmstatus.o:  ftmstatus.c header.h
        -indent ${INDENTPARMS} ftmstatus.c
        cc -c ftmstatus.c

ftmwhere.o:  ftmwhere.c header.h
        -indent ${INDENTPARMS} ftmwhere.c
        cc -c ftmwhere.c

getmsgs.o:  getmsgs.c header.h
        -indent ${INDENTPARMS} getmsgs.c
        cc -c getmsgs.c

hndladvf.o:  hndladvf.c header.h
        -indent ${INDENTPARMS} hndladvf.c
        cc -c hndladvf.c

hndladvi.o:  hndladvi.c header.h
        -indent ${INDENTPARMS} hndladvi.c
        cc -c hndladvi.c

hndladvok.o:  hndladvok.c header.h
        -indent ${INDENTPARMS} hndladvok.c
        cc -c hndladvok.c

hndladvoky.o:  hndladvoky.c header.h
        -indent ${INDENTPARMS} hndladvoky.c
        cc -c hndladvoky.c

hndlclose.o:  hndlclose.c header.h
        -indent ${INDENTPARMS} hndlclose.c
        cc -c hndlclose.c

hndlkilln.o:  hndlkilln.c header.h
        -indent ${INDENTPARMS} hndlkilln.c
        cc -c hndlkilln.c

hndlstatus.o:  hndlstatus.c header.h
        -indent ${INDENTPARMS} hndlstatus.c
        cc -c hndlstatus.c
```

```
hndlto.o:  hndlto.c header.h
        -indent ${INDENTPARMS} hndlto.c
        cc -c hndlto.c

hndltook.o:  hndltook.c header.h
        -indent ${INDENTPARMS} hndltook.c
        cc -c hndltook.c

hndlwhere.o:  hndlwhere.c header.h
        -indent ${INDENTPARMS} hndlwhere.c
        cc -c hndlwhere.c

ifget.o:  ifget.c header.h
        -indent ${INDENTPARMS} ifget.c
        cc -c ifget.c

msgdump.o:  msgdump.c header.h
        -indent ${INDENTPARMS} msgdump.c
        cc -c msgdump.c

msgfrmlist.o:  msgfrmlist.c header.h
        -indent ${INDENTPARMS} msgfrmlist.c
        cc -c msgfrmlist.c

msghndl.o:  msghndl.c header.h
        -indent ${INDENTPARMS} msghndl.c
        cc -c msghndl.c

msgtolist.o:  msgtolist.c header.h
        -indent ${INDENTPARMS} msgtolist.c
        cc -c msgtolist.c

namfrmlist.o:  namfrmlist.c header.h
        -indent ${INDENTPARMS} namfrmlist.c
        cc -c namfrmlist.c

namlookup.o:  namlookup.c header.h
        -indent ${INDENTPARMS} namlookup.c
        cc -c namlookup.c

namtolist.o:  namtolist.c header.h
        -indent ${INDENTPARMS} namtolist.c
        cc -c namtolist.c

nline.o:  nline.c
        -indent ${INDENTPARMS} nline.c
        cc -c nline.c

pick_bkup.o:  pick_bkup.c header.h
        -indent ${INDENTPARMS} pick_bkup.c
        cc -c pick_bkup.c

recv_msg.o:  recv_msg.c header.h
        -indent ${INDENTPARMS} recv_msg.c
        cc -c recv_msg.c

recv_pkt.o:  recv_pkt.c header.h
        -indent ${INDENTPARMS} recv_pkt.c
        cc -c recv_pkt.c

rundaemon.o:  rundaemon.c  /usr/include/signal.h     /usr/include/stdio.h
                           /usr/include/sys/param.h  /usr/include/sys/file.h
```

```
                                /usr/include/sys/ioctl.h
        -indent ${INDENTPARMS} rundaemon.c
        cc -c rundaemon.c

send_msg.o:  send_msg.c header.h
        -indent ${INDENTPARMS} send_msg.c
        cc -c send_msg.c

setcap.o:  setcap.c
        -indent ${INDENTPARMS} setcap.c
        cc -c setcap.c

strlcpy.o:  strlcpy.c
        -indent ${INDENTPARMS} strlcpy.c
        cc -c strlcpy.c

strnlen.o:  strnlen.c
        -indent ${INDENTPARMS} strnlen.c
        cc -c strnlen.c

to.o:   to.c header.h
        -indent ${INDENTPARMS} to.c
        cc -c to.c

zadvclose.o:  zadvclose.c header.h
        -indent ${INDENTPARMS} zadvclose.c
        cc -c zadvclose.c

zdummy.o:  zdummy.c header.h
        -indent ${INDENTPARMS} zdummy.c
        cc -c zdummy.c

zflop.o:  zflop.c header.h
        -indent ${INDENTPARMS} zflop.c
        cc -c zflop.c

zstatus.o:  zstatus.c header.h
        -indent ${INDENTPARMS} zstatus.c
        cc -c zstatus.c

ztest2.o:  ztest2.c  /usr/include/stdio.h  /usr/include/errno.h
                     /usr/include/string.h
        -indent ${INDENTPARMS} ztest2.c
        cc -c ztest2.c

ztest4.o:  ztest4.c  /usr/include/stdio.h  /usr/include/errno.h
                     /usr/include/string.h
        -indent ${INDENTPARMS} ztest4.c
        cc -c ztest4.c

ztestc.o:  ztestc.c  /usr/include/stdio.h  /usr/include/errno.h
                     /usr/include/string.h
        -indent ${INDENTPARMS} ztestc.c
        cc -c ztestc.c

ztestd.o:  ztestd.c  /usr/include/stdio.h  /usr/include/errno.h
                     /usr/include/string.h
        -indent ${INDENTPARMS} ztestd.c
        cc -c ztestd.c

zto.o:  zto.c  header.h
        -indent ${INDENTPARMS} zto.c
```

```
        cc -c zto.c

zwhere.o:  zwhere.c header.h
        -indent ${INDENTPARMS} zwhere.c
        cc -c zwhere.c

zzdummy_in.o:  zzdummy_in.c  /usr/include/stdio.h      /usr/include/sys/socket.h
                             /usr/include/sys/types.h  /usr/include/netinet/in.h
        -indent ${INDENTPARMS} zzdummy_in.c
        cc -c zzdummy_in.c

zzdummy_un.o:  zzdummy_un.c  /usr/include/stdio.h      /usr/include/sys/socket.h
                             /usr/include/sys/types.h  /usr/include/sys/un.h
        -indent ${INDENTPARMS} zzdummy_un.c
        cc -c zzdummy_un.c

zzto_in.o:  zzto_in.c  /usr/include/stdio.h      /usr/include/sys/socket.h
                       /usr/include/sys/types.h  /usr/include/netinet/in.h
                       /usr/include/netdb.h      /usr/include/math.h
                       /usr/include/sys/timeb.h
        -indent ${INDENTPARMS} zzto_in.c
        cc -c zzto_in.c

zzto_un.o:  zzto_un.c  /usr/include/stdio.h      /usr/include/sys/socket.h
                       /usr/include/sys/types.h  /usr/include/sys/un.h
                       /usr/include/math.h       /usr/include/sys/timeb.h
        -indent ${INDENTPARMS} zzto_un.c
        cc -c zzto_un.c


#*****************************************************************************/
#                                                                             /
#   Lint                                                                      /
#                                                                             /
#   Call:  make -f Lint                                                       /
#                                                                             /
#   Purpose - run "lint" against the FTM and the Intercept Library. Standard  /
#             UNIX utility "lint" finds inconsistencies which the C compiler  /
#             misses. In fact, it finds lots of inconsistencies that you      /
#             normally do not care about, so be prepared for a large volume   /
#             of output. But, running "lint" may be invaluable for certain    /
#             types of errors.                                                /
#                                                                             /
#*****************************************************************************/

FTMSRCS = conall.c contimout.c ifget.c ffget.c findbuf.c ftmisdown.c hndladvf.c
          hndladvi.c hndladvok.c hndladvoky.c hndlclose.c hndlkilln.c
          hndlstatus.c hndlto.c hndltook.c hndlwhere.c msgdump.c msgfrmlist.c
          msghndl.c msgtolist.c namfrmlist.c namlookup.c namtolist.c nline.c
          pick_bkup.c recv_msg.c recv_pkt.c send_msg.c setcap.c strlcpy.c

INTSRCS = adv.c copycf.c findbuf.c from.c ftmclose.c ftmstatus.c ftmwhere.c
          getmsgs.c msgfrmlist.c msgtolist.c recv_msg.c recv_pkt.c send_msg.c
```

```
          setcap.c strlcpy.c strnlen.c to.c

all:       lint

lint:      ftm.c ftmkill.c ftmstart.c rundaemon.c ztest2.c llib-lftmlb.ln
          llib-lintlb.ln
        lint ftm.c llib-lftmlb.ln
        lint ftmkill.c llib-lftmlb.ln
        lint ftmstart.c llib-lftmlb.ln
        lint rundaemon.c llib-lftmlb.ln
        lint ztest2.c llib-lintlb.ln

llib-lftmlb.ln: $(FTMSRCS) header.h
        lint -Cftmlb $(FTMSRCS)

llib-lintlb.ln: $(INTSRCS) header.h
        lint -Cintlb $(INTSRCS)
```

```
/*****************************************************************************/
/*                                                                           */
/*   ftmbkup.lis                                                             */
/*                                                                           */
/*   Purpose - equate advertised names with the complete path name to the    */
/*             advertising program. This is used for program restarts after  */
/*             task and node failures.                                       */
/*                                                                           */
/*****************************************************************************/

EACH LINE OF THIS FILE MUST BE FORMATTED AS FOLLOWS:
   FIELD 1 : ADVERTISED NAME
   FIELD 2 : PROGRAM COMPLETE PATH NAME

   FIELD 1 STARTS IN COLUMN 1.
   THIS IS FOLLOWED BY AT LEAST 1 TAB.
   THIS IS FOLLOWED BY FIELD 2.
   THIS IS FOLLOWED BY A NEWLINE OR EOF.

ANY LINES NOT FOLLOWING THIS EXACT FORMAT ARE IGNORED, WHICH IS WHY THESE
COMMENT LINES CAN APPEAR HERE. THAT IS, ANY LINE WITH NO TABS IS CONSIDERED
TO BE A COMMENT LINE.

AN EXAMPLE OF A REAL DATA LINE FOLLOWS.  NOTE THAT IT WILL NOT HURT TO LEAVE THE
LINE IN THE FILE, AS LONG AS YOUR PROGRAMS DO NOT USE MY SILLY ADVERTISED NAME.
THERE IS ONE TAB AFTER THE 'Gazorp':

Gazorp          /home/auvschi/jun88/testsw

FOLLOWING ARE THE ACTUAL DATA LINES FOR DENIS'S TEST PROGRAMS:

WORLD           /home/auvschi/apps/gworld
SKIPPER         /home/auvschi/apps/gskip
NAVIGATOR       /home/auvschi/apps/gnav
```

```
WEAPON          /home/auvschi/apps/gweap

FOLLOWING ARE THE ACTUAL DATA LINE FOR THE RELOCATION TEST PROGRAM "ZFLOP.C"
(RECALL THAT THE FTM SYSTEM IS CASE_INSENSITIVE TO ADVERTISED NAMES):

flop            /u/auvusersl/auvschi/ftm/zflop
```

```
*****************************************************************************
**                                                                         **
**   ftms.lis                                                              **
**                                                                         **
**   Purpose - lists the systems on which ftms must initially be running   **
**             to create the fault tolerant environment.                   **
**                                                                         **
**   Notes - Comment lines must begin with '*'                             **
**         - Null lines are ignored                                        **
**                                                                         **
*****************************************************************************

node1
node2
node3
node4
node5
node6
node7
node8
node9
node10
node11
node12
node13
node14
node15
node16
```

```
/*****************************************************************************
-                                                                           -
-    rkill                                                                  -
-                                                                           -
-    Purpose - this shell script sends signals to remote processes.         -
-              Given a signal type (say, for example: TERM) and a program   -
-              name as called (say, for example: /u/auvusers1/auvschi/ftm/ftm) -
-              rkill would be called as:                                    -
-                                                                           -
-                   rkill  -TERM  /u/auvusers1/auvschi/ftm/ftm              -
-                                                                           -
-              A complete list of signal types can be found in the manual   -
-              pages under the function: signal().                          -
-                                                                           -
-    Note - this shell script was written by David Hess. It and routine     -
-           rundaemon.c are the only code in Appendix B not written by the  -
-           author of this dissertation.                                    -
-                                                                           -
-****************************************************************************/

#!/bin/sh
if (test $# -lt 2) then
        echo "Usage: rkill signal process_name"
        exit
fi
for NODE in auvc1 auvc2 auvc3 auvc4 auvc5 auvc6 auvc7 auvc8 auvc9 auvc10 \
                auvc11 auvc12 auvc13 auvc14 auvc15 auvc16
do
        if (ping $NODE 2 > /dev/null 2>&1) then
                temp=`rsh $NODE "ps -ax | grep '[0-9] $2'"`
                temp1=`echo $temp | awk '{ print $1 }'`

                if (test -z "$temp") then
                        echo "[$NODE]"
                        echo "not found"
                else
                        echo "[$NODE]"
                        echo "$temp"
                        rsh $NODE kill $1 $temp1
                fi
        else
                echo Node $NODE appears to be down
        fi
        echo
done
```

```
/*****************************************************************************
-                                                                            -
-    header.h                                                                -
-                                                                            -
-    Purpose - miscellaneous #includes and global constants.                 -
-                                                                            -
-****************************************************************************/

#include <errno.h>
#include <fcntl.h>
#include <malloc.h>
#include <math.h>
#include <netdb.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/file.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/timeb.h>
#include <sys/un.h>
#include <sys/wait.h>

/* ------------------------------------------------------------------------ */

/* NOTE - FDEBUG AND IDEBUG AND MDEBUG USE stdout - IF THE FTMS ARE STARTED
 *        BY THE SUPPLIED FTMSTART UTILITY, stdout (and stderr) ARE REDIRECTED
 *        TO SOCKPATH/stdout_xxxxxx (and SOCKPATH/stderr_xxxxxx), WHERE
 *        xxxxxx IS THE HOSTNAME OF THE SYSTEM ON WHICH THE INDIVIDUAL FTM
 *        IS RUNNING.                                                        */

#define FDEBUG            0     /* INCLUDE DEBUG CODE IN FTM                 */
#define IDEBUG            0     /* INCLUDE DEBUG CODE IN INTERCEPT LIB       */
#define MDEBUG            0     /* INCLUDE DEBUG CODE IN FTM AND INTERCEPT
                                 * LIB TO REPORT ON ALL MESSAGE TRANSFERS AT
                                 * SEND AND AT RECV TIMES.                   */
#define FDEBUGLOOP        2     /* DEBUG TABLE PRINTOUT INITIAL VALUE. USES
                                 * STATIC VARIABLE yesno IN ftm.c. ONLY USED
                                 * IF FDEBUG != 0.
                                 * 0 => DO NOT PRINT TABLES WITH EACH FTM LOOP
                                 * 1 => PRINT TABLES AND STOP FOR STDIN
                                 *      CONFIRMATION WITH EACH FTM LOOP
                                 *      IN THIS CASE THE FTM MUST BE RUN IN
                                 *      THE FOREGROUND.
                                 * 2 => PRINT TABLES WITH EACH FTM LOOP AND
                                 *      CONTINUE. IN THIS CASE, MAKE THE
                                 *      #define FTM_SLEEP_TIME A LARGE VALUE  */

#define FTMPORT           8877  /* THIS PORT MUST BE AVAULABLE ON ALL SYSTEMS */
#define FTMPATH          "/u/auvusersl/auvschi/ftm/"
                                 /* FTMPATH is the path to the FTM modules.
                                 * This is usually file served between nodes
                                 * in the FTM system.                        */
#define SOCKPATH         "/tmp/auvschi/ftm/"
                                 /* SOCKPATH is the path used by the FTM
                                 * system for UNIX sockets and debugging
                                 * output. This is usually local to each
                                 * node for performance, especially in debug
```

```
                                 * mode. WARNING - If this directory is not
                                 * accessible the FTM system will not work and
                                 * may not be able to tell you why it fails.  */

/* ------------------------------------------------------------------------ */

#define printe(x1);           fprintf (stderr, x1);
#define printe2(x1,x2);       fprintf (stderr, x1,x2);
#define printe3(x1,x2,x3);    fprintf (stderr, x1,x2,x3);
#define printe4(x1,x2,x3,x4); \
                              fprintf (stderr, x1,x2,x3,x4);
#define printe9(x1,x2,x3,x4,x5,x6,x7,x8,x9);  \
                              fprintf (stderr, x1,x2,x3,x4,x5,x6,x7,x8,x9);

#define FTMFILE           "ftms.lis"
#define BKUPFILE          "ftmbkup.lis"
#define FTMPROG           "ftm"
#define TIMEOUTPROG       "timeout"
#define DAEMONPROG        "rundaemon"
#define MAXNAMELEN             20 /* MAX LENGTH OF ADVERTISED NAME           */
#define FTM_WAIT_CONNECT       60 /* MAX TIME FOR FTMS TO CONNECT TO EACH
                                   * OTHER (SEC)                             */
#define FTM_SLEEP_CONNECT   50000 /* MAX TIME FOR AN FTM TO SLEEP BETWEEN
                                   * TRIES TO CONNECT TO ANOTHER FTM
                                   * (micro-SEC)                             */
#define FTM_SLEEP_TIME    2000000 /* FTM SLEEP TIME BEFORE A LOOP IS FORCED
                                   * SO AS TO ASSURE NO LOST INTERRUPT AND TO
                                   * CHECK FOR CONNECTED FTM AND APPLICATION
                                   * ABNORMAL TEPMINATIONS (micro-SEC)       */
#define IC_WAIT_CONNECT        10 /* MAX TIME FOR IC TO CONNECT TO LOCAL FTM
                                   * (SEC)                                   */
#define IC_SLEEP_CONNECT    25000 /* MAX TIME FOR AN IC TO SLEEP BETWEEN
                                   * TRIES TO CONNECT TO LOCAL FTM
                                   * (micro-SEC)                             */
#define IC_SLEEP_TIME     2000000 /* IC SLEEP TIME BEFORE A LOOP IS FORCED SO
                                   * AS TO ASSURE NO LOST INTERRUPT
                                   * (micro-SEC)                             */
#define MSG2_SLEEP         250000 /* SLEEP TIME WHILE WAITING FOR REMAINDER
                                   * OF MSG TO ASSURE NO LOST INTERRUPT
                                   * (micro-SEC)                             */
#define MSG2_WAIT              10 /* MAX TIME BETWEEN RECEIPT OF SECTIONS OF
                                   * 1 MSG BEFORE SOCKET ASSUMED DOWN
                                   * (SEC)                                   */
#define MSG_LIMIT              10 /* IMMEDIATELY INTERRUPT ICs IF THIS MANY
                                   * MESSAGES HAVE BEEN SENT TO THE CONNECTED
                                   * APPLICATIONS SINCE LAST INTERRUPT       */
#define PACKETSIZE           1024 /* SIZE OF PACKET (BYTES). THE VALUE (1024)
                                   * APPEARS TO BE OPTIMAL ON OUR SUN SPARC
                                   * ETHERNET BASED NETWORK. IF ZERO, MSG
                                   * PORTION OF MESSAGES WILL NOT BE PADDED.  */
#define MAXMSGLEN            2048 /* MAX SIZE OF USER MSG (BYTES).           */
#define TAB                  0x09
#define NL                   0x0A
#define LOOP          for ( ; ; )

extern void     contimout();
extern void     getmsgs();
extern void     msgdump();
extern void     nline();
extern void     setcap();
extern char    *strlcpy();
```

```
struct ftm_time {                          /* TO PASS TIMEOUT VALUES THROUGHOUT */
   struct timeval  timeval_perm,           /* THE THE FTM & IC ROUTINES         */
                   timeval_temp,
                   msgtime_perm,
                   msgtime_temp;
   fd_set          readfds_perm,
                   readfds_temp;
   int             widthfds;
};

struct ftmtable {                          /* ENTRY FOR A NODE IN FTM SYSTEM    */
   char            mach[MAXHOSTNAMELEN + 1];  /* node name                       */
   u_long          addr;                   /* internet address                  */
   int             sock;                   /* logical socket number             */
};

struct namtable {                          /* ENTRY FOR AN ADVERTISED NAME      */
   char            name[MAXNAMELEN + 1];   /* advertised name                   */
   char            status;                 /* connection status of application */
   int             lsock;                  /* socket number from local ftm      */
   int             fsock;                  /* sck num: this ftm to local ftm    */
   int             pid;                    /* process id of application         */
   int             bkupftm;                /* logical socket of backup ftm      */
   int             msgs;                   /* num of msgs since last signal     */
   struct namtable *next;                  /* ptr to next namtable entry        */
   struct namtable *prev;                  /* ptr to prev namtable entry        */
};

struct msgbuf {                                  /* ENTRY FOR A MSG IN MSG LIST */
   struct msgbuf  *next;                         /* ptr to next msgbuf entry    */
   struct msgbuf  *prev;                         /* ptr to prev msgbuf entry    */
   char            cmd[7];                       /* message type                */
   char            from;                         /* message from FTM or IC ?    */
   char            status[6];                    /* action taken on msg so far  */
   char            name_to[MAXNAMELEN + 1];      /* target of message           */
   char            name_from[MAXNAMELEN + 1];    /* originator of message       */
   char            seqnum[12];                   /* message sequence number     */
   char            errno[12];                    /* error code for exceptions   */
   char            mlen[12];                     /* length of the next field    */
   char            msg[MAXMSGLEN + 1];           /* data portion of the message */
};
#define MSGBUF_HDR        8
#define MSGBUF_MAX       (sizeof (struct msgbuf))
#define MSGBUF_MIN       (MSGBUF_MAX - MAXMSGLEN)
```

```
/*****************************************************************************
-                                                                            -
-    adv.c                                                                   -
-                                                                            -
-    Purpose - called by application program to advertise the name by which -
-              the program will be known throughout the system.              -
-                                                                            -
-****************************************************************************/

#include          "header.h"

struct msgbuf  *msgfirst,
               *msglast;

char            intadvyet = 'N',
                intmyname[MAXNAMELEN + 1],
                ftm_ic = 'I';

int             ftmsock,
                intseqnum,
                intpidme = -1,
                int_msgs;       /* ----- DUMMY FOR EXTERN IN SEND_MSG ----- */

int             namlookup()
{                               /* ----- DUMMY FOR EXTERN IN SEND_MSG ----- */
   ;
}

struct ftm_time ftmtime;

static char     canrepeat;


int             adv(name)
   char           *name;

{
   int             rc,
                   err,
                   oldmask,
                   advx();


   if ((intpidme != getpid()) || (intadvyet == 'R')) {
      msgfirst = NULL;
      msglast = NULL;
      intadvyet = 'N';
      ftmsock = -2;
      intseqnum = 1;
      intpidme = getpid();
      canrepeat = 'Y';
   }
   if (intadvyet == 'R')
      intadvyet = 'N';

   if (signal(SIGPIPE, SIG_IGN) == BADSIG) {
      perror("ADV: SIGPIPE SIGNAL");
      errno = EFAULT;
      return (-1);
   }
   if (signal(SIGUSR1, getmsgs) == BADSIG) {
      perror("ADV: SIGUSR1 SIGNAL");
```

```
        errno = EFAULT;
        return (-1);
    }
    oldmask = sigblock(sigmask(SIGUSR1));
    rc = advx(name);
    err = errno;
    sigsetmask(oldmask);
    errno = err;
    return (rc);
}
static int      advx(name)
    char            *name;

{
    struct sockaddr_un server;

    struct msgbuf  *mptr,
                   *mptr2;

    int             wait,
                    len;

    char            hbuf[MAXHOSTNAMELEN + 1],
                    fname[MAXPATHLEN + 1],
                    capname[MAXNAMELEN + 1];


    if (strnlen(name, 1) == 0) {
        errno = EINVAL;
        return (-1);
    }
    if ((strnlen(name, 2) == 1) && (name[0] == '*')) {
        errno = EINVAL;
        return (-1);
    }
    if (intadvyet == 'Y') {
        errno = EALREADY;
        return (-1);
    }
    if (canrepeat == 'N') {
        printe("ADV: PREVIOUS ERROR PRECLUDES REPEAT CALL\n");
        errno = EFAULT;
        return (-1);
    }
    bzero(capname, sizeof(capname));
    strlcpy(capname, name, sizeof(capname) - 1);
    setcap(capname, sizeof(capname) - 1);

#if IDEBUG
    printf("DEBUG: ADV: CAPNAME = %s\n", capname);
#endif

    bzero(hbuf, sizeof(hbuf));
    if (gethostname(hbuf, sizeof(hbuf) - 1) < 0) {
        perror("ADV: GETHOSTNAME");
        errno = EFAULT;
        return (-1);
    }
    bzero(fname, sizeof(fname));
    strcpy(fname, SOCKPATH);
    strcat(fname, FTMPROG);
    strcat(fname, hbuf);
```

```
   strcat(fname, ".sck");

#if IDEBUG
   printf("DEBUG: ADV: FNAME = %s\n", fname);
#endif

   wait = 0;
   LOOP {
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, fname);

      if ((ftmsock = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
         perror("ADV: SOCKET");
         return (-1);
      }
      if (connect(ftmsock, &server, strlen(server.sun_path) + 2) < 0) {
         if ((errno == EWOULDBLOCK) || (errno == ECONNREFUSED)
             || (errno == ENOENT)) {
            if (wait * IC_SLEEP_CONNECT > IC_WAIT_CONNECT * 1000000) {
               printe("ADV: EXCESSIVE WAIT TO CONNECT TO FTM\n");
               close(ftmsock);
               errno = EFAULT;
               return (-1);
            } else {
               close(ftmsock);
               usleep(IC_SLEEP_CONNECT);
               ++wait;
               continue;
            }
         } else {
            perror("ADV: CONNECT");
            close(ftmsock);
            errno = EFAULT;
            return (-1);
         }
      } else
         break;
   }

   if (fcntl(ftmsock, F_SETFL, FNDELAY) < 0) {
      perror("ADV: FCNTL");
      close(ftmsock);
      errno = EFAULT;
      return (-1);
   }
   if (setsockopt(ftmsock, SOL_SOCKET, SO_KEEPALIVE, (char *) 0, 0) < 0) {
      perror("ADV: SETSOCKOPT-1");
      errno = EFAULT;
      return (-1);
   }
   if (setsockopt(ftmsock, SOL_SOCKET, SO_DONTLINGER, (char *) 0, 0) < 0) {
      perror("ADV: SETSOCKOPT-2");
      errno = EFAULT;
      return (-1);
   }
   if (((mptr = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL)
       || ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL)) {
      perror("ADV: MALLOC");
      if (mptr != NULL)
         free((char *) mptr);
      close(ftmsock);
      errno = EFAULT;
```

```
        return (-1);
    }
    bzero((char *) mptr, MSGBUF_MAX);
    bzero((char *) mptr2, MSGBUF_MAX);

    strcpy(mptr->cmd, "ADV");
    mptr->from = 'I';
    sprintf(mptr->status, "%d", -1);
    strcpy(mptr->name_to, "");
    strcpy(mptr->name_from, capname);
    sprintf(mptr->seqnum, "%d", intseqnum);
    ++intseqnum;
    strcpy(mptr->errno, "0");
    sprintf(mptr->msg, "%d", getpid());
    sprintf(mptr->mlen, "%d", strlen(mptr->msg));

    len = MSGBUF_MIN - MSGBUF_HDR + strlen(mptr->msg);
    if ((send_msg(ftmsock, mptr->cmd, len)) < 0) {
        free((char *) mptr);
        free((char *) mptr2);
        close(ftmsock);
        perror("ADV: SEND_MSG");
        errno = EFAULT;
        return (-1);
    }
    canrepeat = 'N';

#if IDEBUG
    printf("DEBUG: ADV: THE ADV MESSAGE HAS BEEN SENT TO LOCAL FTM = %d\n",
            ftmsock);
    printf("DEBUG: ADV: NOW WAIT FOR THE ADVOK MESSAGE FROM THE FTM\n");
#endif

    ftmtime.widthfds = getdtablesize();
    FD_ZERO(&ftmtime.readfds_perm);
    FD_SET(ftmsock, &ftmtime.readfds_perm);
    ftmtime.timeval_perm.tv_sec = IC_SLEEP_TIME / 1000000;
    ftmtime.timeval_perm.tv_usec = IC_SLEEP_TIME % 1000000;
    ftmtime.msgtime_perm.tv_sec = MSG2_SLEEP / 1000000;
    ftmtime.msgtime_perm.tv_usec = MSG2_SLEEP % 1000000;

 /*
  * LOOP UNTIL THE ADVOK MESSAGE IS RECEIVED
  */

    LOOP {

        getmsgs(-1);

        if (ftmsock < 0) {
            errno = ENOTCONN;
            return (-1);
        }
        if (findbuf("ADVNO", "", mptr2) < 0) {
            if (errno != EWOULDBLOCK) {
                printe("ADV: BAD RETURN FROM FINDBUF-1\n");
                free((char *) mptr);
                free((char *) mptr2);
                close(ftmsock);
                errno = EFAULT;
                return (-1);
            } else;
```

```
      } else {
         sscanf(mptr2->errno, "%d", &errno);
         free((char *) mptr);
         free((char *) mptr2);
         return (-1);
      }

      if (findbuf("ADVOK", "", mptr2) < 0) {
         if (errno != EWOULDBLOCK) {
            printe("ADV: BAD RETURN FROM FINDBUF-2\n");
            free((char *) mptr);
            free((char *) mptr2);
            close(ftmsock);
            errno = EFAULT;
            return (-1);
         } else;
      } else if (strcmp(mptr2->name_to, capname) != 0) {
         printe2("ADV: ADVOK NAME CHANGED TO %s\n", mptr2->name_to);
         free((char *) mptr);
         free((char *) mptr2);
         close(ftmsock);
         errno = EFAULT;
         return (-1);
      } else if (strcmp(mptr->seqnum, mptr2->seqnum) != 0) {
         printe3("ADV: SEQNUM MISMATCH %s %s\n", mptr->seqnum, mptr2->seqnum);
         free((char *) mptr);
         free((char *) mptr2);
         close(ftmsock);
         errno = EFAULT;
         return (-1);
      } else
         break;
      bcopy((char *) &ftmtime.readfds_perm, (char *) &ftmtime.readfds_temp,
            sizeof(struct fd_set));
      bcopy((char *) &ftmtime.timeval_perm, (char *) &ftmtime.timeval_temp,
            sizeof(struct timeval));
      select(ftmtime.widthfds, &ftmtime.readfds_temp, NULL, NULL,
             &ftmtime.timeval_temp);
   }

#if IDEBUG
   printf("DEBUG: ADV: THE ADVOK MESSAGE HAS BEEN RECEIVED\n");
#endif

   strcpy(intmyname, capname);
   intadvyet = 'Y';
   free((char *) mptr);
   free((char *) mptr2);
   return (0);
}
```

```
/*****************************************************************************
-                                                                            -
-    conall.c                                                                -
-                                                                            -
-    Purpose - called by each FTM as part of initialization. Causes the FTMs -
-              to allow 0-hop communication through connected sockets.       -
-                                                                            -
-****************************************************************************/

#include            "header.h"

int             conall()
{
   struct sockaddr_in server,
                      client;

   struct hostent *hp;

   FILE           *filex;

   int             addrme,
                   i,
                   i2,
                   s1,
                   s2,
                   len;

   char            buf[MAXHOSTNAMELEN + 1],
                   fname[MAXPATHLEN + 1],
                   found;

   struct ftmtable *ftmptr2;


   extern struct ftmtable *ftmptr;
   extern int      ftmsize,
                   ftmme;


   if (gethostname(buf, sizeof(buf)) < 0) {
      perror("CONALL: GETHOSTNAME");
      return (-1);
   }
   buf[sizeof(buf) - 1] = 0x00;
   if ((hp = gethostbyname(buf)) == NULL) {
      perror("CONALL: GETHOSTBYNAME-1");
      printe2("CONALL: COULD NOT LOCATE MACHINE NAME: %s\n", buf);
      return (-1);
   } else if (hp->h_length != 4) {
      printe2("CONALL: ILLEGAL NET ADDR LENGTH-1 = %d\n", hp->h_length);
      return (-1);
   } else if (hp->h_addrtype != AF_INET) {
      printe2("CONALL: ILLEGAL ADDRTYPE-1 = %d\n", hp->h_addrtype);
      return (-1);
   } else
      addrme = *(int *) hp->h_addr;

   bzero(fname, sizeof(fname));
   strcpy(fname, FTMPATH);
   strcat(fname, FTMFILE);
   if ((filex = fopen(fname, "r")) == NULL) {
      perror("CONALL: OPEN");
```

```
   return (-1);
}
ftmsize = 0;
while (fgets(buf, sizeof(buf), filex) != NULL) {
   if (buf[0] == '\n')
      continue;
   if (buf[0] == '*') {
      while (buf[strlen(buf) - 1] != '\n')
         if (fgets(buf, sizeof(buf), filex) == NULL) {
            printe2("CONALL: ERROR DURING FIRST READ OF %s\n", FTMFILE);
            return (-1);
         }
      continue;
   }
   ++ftmsize;
}
if (ftmsize < 1) {
   printe2("CONALL: FILE ftms.lis SIZE = %d NOT ACCEPTABLE\n", ftmsize);
   return (-1);
}
if ((ftmptr = (struct ftmtable *) malloc(ftmsize * sizeof(*ftmptr)))
    == NULL) {
   perror("CONALL: MALLOC");
   return (-1);
} else
   bzero((char *) ftmptr, ftmsize * sizeof(*ftmptr));

rewind(filex);

for (ftmptr2 = ftmptr; ftmptr2 < ftmptr + ftmsize; ++ftmptr2) {
   while (fgets(buf, sizeof(buf), filex) != NULL) {
      if (buf[0] == '\n')
         continue;
      if (buf[0] == '*') {
         while (buf[strlen(buf) - 1] != '\n')
            if (fgets(buf, sizeof(buf), filex) == NULL) {
               printe2("CONALL: ERROR DURING SECOND READ OF %s\n", FTMFILE);
               return (-1);
            }
         continue;
      }
      if ((buf[0] == '*') || (buf[0] == '\n'))
         continue;
      nline(buf, sizeof(buf));
      if (strcmp(buf, "cssun0") == 0)
         strcpy(buf, "cssun");
      strcpy(ftmptr2->mach, buf);
      ftmptr2->sock = -2;
      break;
   }
   if (ftmptr2->sock == 0) {
      perror("CONALL: READ");
      return (-1);
   }
}

ftmme = -1;
for (i = 0; i < ftmsize; ++i) {
   if ((hp = gethostbyname((ftmptr + i)->mach)) == NULL) {
      perror("CONALL: GETHOSTBYNAME-2");
      printe2("CONALL: COULD NOT LOCATE MACHINE NAME: %s\n",
              (ftmptr + i)->mach);
```

```
            return (-1);
         } else if (hp->h_length != 4) {
            printe2("CONALL: ILLEGAL NET ADDR LENGTH-2 = %d\n", hp->h_length);
            return (-1);
         } else if (hp->h_addrtype != AF_INET) {
            printe2("CONALL: ILLEGAL ADDRTYPE-2 = %d\n", hp->h_addrtype);
            return (-1);
         } else {
            bcopy(hp->h_addr, (char *) &(ftmptr + i)->addr, 4);
            if (strlen(hp->h_name) > MAXHOSTNAMELEN) {
               printe2("CONALL: MACH NAME %s IS TOO LONG\n", hp->h_name);
               return (-1);
            }
            strcpy((ftmptr + i)->mach, hp->h_name);
            if (*(int *) hp->h_addr == addrme) {
               if (ftmme != -1) {
                  printe2("CONALL: HOST: %s IS LISTED TWICE IN ftms.lis FILE\n",
                          (ftmptr + i)->mach);
                  return (-1);
               } else {
                  (ftmptr + i)->sock = -1;
                  ftmme = i;
               }
            }
         }
      }

      if (ftmme == -1) {
         printe("CONALL: THIS HOST NOT FOUND IN FILE\n");
         return (-1);
      }
      if (fclose(filex) != 0) {
         perror("CONALL: CLOSE FILE");
         return (-1);
      }
      if ((s1 = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
         perror("CONALL: SOCKET-1");
         return (-1);
      }
      if (signal(SIGALRM, contimout) == BADSIG) {
         perror("CONALL: SIGNAL-1");
         return (-1);
      }
      alarm(FTM_WAIT_CONNECT);
      bzero((char *) &server, sizeof(server));
      server.sin_port = FTMPORT;
      if (bind(s1, (struct sockaddr *) & server, sizeof(server)) < 0) {
         perror("CONALL: BIND");
         printe2("CONALL: BIND ADDR = %d\n", FTMPORT);
         return (-1);
      }
      if (listen(s1, 5) < 0) {
         perror("CONALL: LISTEN");
         return (-1);
      }
      for (i = 0; i < ftmme; ++i) {
         bzero((char *) &client, sizeof(client));
         len = sizeof(client);
         if ((s2 = accept(s1, &client, &len)) < 0) {
            perror("CONALL: ACCEPT");
            return (-1);
         } else if (fcntl(s2, F_SETFL, FNDELAY) < 0) {
```

```
            perror("CONALL: FCNTL-1");
            return (-1);
         } else if (setsockopt(s2, SOL_SOCKET, SO_KEEPALIVE, (char *) 0, 0) < 0) {
            perror("CONALL: SETSOCKOPT-1");
            return (-1);
         } else if (setsockopt(s2, SOL_SOCKET, SO_DONTLINGER, (char *) 0, 0) < 0) {
            perror("CONALL: SETSOCKOPT-2");
            return (-1);
         } else {
            found = 'N';
            for (i2 = 0; i2 < ftmme; ++i2) {
               if ((ftmptr + i2)->addr == *(u_long *) & client.sin_addr) {
                  if ((ftmptr + i2)->sock != -2) {
                     printe("CONALL: DUPLICATE TABLE ENTRY FOUND\n");
                     return (-1);
                  } else {
                     (ftmptr + i2)->sock = s2;
                     found = 'Y';
                     break;
                  }
               }
            }

#if FDEBUG
            printf("DEBUG: CONALL: ACCEPT FOR MACHINE = %s\n",
                   (ftmptr + i2)->mach);
#endif
         }
         if (found == 'N') {
            printe2("CONALL: ENTRY NOT FOUND IN TABLE - ADDR: %X\n",
                    *(u_long *) & client.sin_addr);
            return (-1);
         }
      }

      if (close(s1) < 0) {
         perror("CONALL: CLOSE SOCKET");
         return (-1);
      }
      for (i = ftmme + 1; i < ftmsize; ++i) {
         LOOP {
            if ((s1 = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
               perror("CONALL: SOCKET-2");
               return (-1);
            } else {
               bzero((char *) &server, sizeof(server));
               server.sin_family = AF_INET;
               server.sin_port = FTMPORT;
               bcopy((char *) &(ftmptr + i)->addr, (char *) &server.sin_addr, 4);
               if (connect(s1, &server, sizeof(server)) < 0)
                  if ((errno != ETIMEDOUT) && (errno != ECONNREFUSED)) {
                     perror("CONALL: CONNECT");
                     return (-1);
                  } else {
                     close(s1);
                     usleep(FTM_SLEEP_CONNECT);
                     continue;
                  }
               if (fcntl(s1, F_SETFL, FNDELAY) < 0) {
                  perror("CONALL: FCNTL-2");
                  return (-1);
               } else if (setsockopt
```

```
                    (sl, SOL_SOCKET, SO_KEEPALIVE, (char *) 0, 0) < 0) {
            perror("CONALL: SETSOCKOPT-3");
            return (-1);
         } else if (setsockopt(sl, SOL_SOCKET, SO_DONTLINGER, (char *) 0, 0)
                    < 0) {
            perror("CONALL: SETSOCKOPT-4");
            return (-1);
         } else
            break;
      }
   }
   (ftmptr + i)->sock = sl;

#if FDEBUG
   printf("DEBUG: CONALL: CONNECT TO MACHINE = %s\n", (ftmptr + i)->mach);
#endif
   }

   alarm(0);
   if (signal(SIGALRM, SIG_IGN) == BADSIG) {
      perror("CONALL: SIGNAL-2");
      return (-1);
   }
   return (0);
}
```

```
/*****************************************************************************
-                                                                            -
-    contimout.c                                                             -
-                                                                            -
-    Purpose - stub to handle timeouts while FTMs are waiting to connect.    -
-                                                                            -
-****************************************************************************/

#include        "header.h"

void            contimout()
{
   printe("CONTIMOUT: FTM TIMED OUT WAITING TO CONNECT\n");
   printe2("CONTIMOUT: FTM_WAIT_CONNECT TIME = %d SECS\n", FTM_WAIT_CONNECT);
   exit(-1);
}
```

```
/*****************************************************************************
-                                                                            -
-    copycf.c                                                                -
-                                                                            -
-    Purpose - called by application program to copy a closed file to the    -
-              current backup system.                                        -
-                                                                            -
-****************************************************************************/

#include        "header.h"

int             copycf(name, filename)
   char           *name,
                  *filename;

{
   char            mach[MAXHOSTNAMELEN + 1],
                   bkup[MAXHOSTNAMELEN + 1],
                   shell_command[sizeof("rcp ") - 1
                                 +              MAXPATHLEN
                                 +              sizeof(" ") - 1
                                 +              MAXHOSTNAMELEN
                                 +              sizeof(":") - 1
                                 +              MAXPATHLEN
                                 +              1];

   if ((strnlen(name, 1) == 0) || (strnlen(filename, 1) == 0)
       || (strnlen(name, MAXNAMELEN + 1) == MAXNAMELEN + 1)
       || (strnlen(filename, MAXPATHLEN + 1) == MAXPATHLEN + 1)) {
      errno = EINVAL;
      return (-1);
   }
   if (ftmwhere(name, mach, bkup) < 0) {
      errno = ENOTCONN;
      return (-1);
   }
   if (bkup[0] == 0x00) {
      errno = EWOULDBLOCK;
      return (-1);
   }
   strcpy(shell_command, "rcp ");
   strcat(shell_command, filename);
   strcat(shell_command, " ");
   strcat(shell_command, bkup);
   strcat(shell_command, ":");
   strcat(shell_command, filename);

#if DEBUG
   printf("DEBUG: COPYCF: NAME = %s, BKUP = %s\n", name, bkup);
   printf("DEBUG: COPYCF: COMMAND = %s\n", shell_command);
#endif

   return (system(shell_command));
}
```

```
/*****************************************************************************
-                                                                            -
-    ffget.c                                                                 -
-                                                                            -
-    Purpose - called by an FTM to gather any pending messages from other    -
-              connected FTMs.                                               -
-                                                                            -
-****************************************************************************/

#include            "header.h"

int             ffget()
{
   int             i,
                   s,
                   lsock,
                   pid,
                   thispass,
                   allpasses,
                   bkupftm;

   struct msgbuf  *mptr2;

   extern struct ftmtable *ftmptr;
   extern int      ftmsize;

   extern struct ftm_time ftmtime;


   if ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL) {
      perror("FFGET: MALLOC");
      errno = EFAULT;
      return (-1);
   }
   allpasses = 0;
   LOOP {
      thispass = 0;
      for (i = 0; i < ftmsize; ++i) {
         s = (ftmptr + i)->sock;
         if (s < -2) {
            if (ftmisdown(ftmptr + i) < 0) {
               printe("FFGET: BAD RETURN FROM FTMISDOWN-1\n");
               free((char *) mptr2);
               errno = EFAULT;
               return (-1);
            } else {
               (ftmptr + i)->sock = -2;
               continue;
            }
         }
         if (s >= 0) {
            bzero(mptr2->cmd, MSGBUF_MAX - MSGBUF_HDR);
            if (recv_msg(s, mptr2->cmd, MSGBUF_MAX - MSGBUF_HDR) < 0) {
               if (errno == EWOULDBLOCK)
                  continue;
               else if (errno == ENOTCONN) {

#if FDEBUG
                  printf("DEBUG: FFGET: SOCKET CLOSED = %d\n", s);
#endif

                  close(s);
```

```
                (ftmptr + i)->sock = -3 - s;
                if (ftmisdown(ftmptr + i) < 0) {
                   printe("FFGET: BAD RETURN FROM FTMISDOWN-2\n");
                   free((char *) mptr2);
                   errno = EFAULT;
                   return (-1);
                } else {
                   (ftmptr + i)->sock = -2;
                   continue;
                }
             } else {
                printe("FFGET: BAD RETURN FROM RECV_MSG\n");
                free((char *) mptr2);
                errno = EFAULT;
                return (-1);
             }
          }

#if FDEBUG
          printf("DEBUG: FFGET: MSG FOUND = %s %c %s %s %s %s %s %s %s\n",
                 mptr2->cmd,
                 mptr2->from,
                 mptr2->status,
                 mptr2->name_to,
                 mptr2->name_from,
                 mptr2->seqnum,
                 mptr2->errno,
                 mptr2->mlen,
                 mptr2->msg);
#endif

          if (strcmp(mptr2->cmd, "ADV") == 0) {
             sscanf(mptr2->msg, "%d %d %d", &pid, &lsock, &bkupftm);
             sprintf(mptr2->msg, "%d %d %d %d", pid, lsock, bkupftm, s);
             sprintf(mptr2->mlen, "%d", strlen(mptr2->msg));
          }
          if (msgtolist(mptr2) < 0) {
             printe("FFGET: BAD RETURN FROM MSGTOLIST\n");
             free((char *) mptr2);
             errno = EFAULT;
             return (-1);
          }
          ++thispass;
          --i;
       }
    }
    allpasses += thispass;
/* ----- ONLY ALLOW ONE MESSAGE PER SOCKET PER FTM MAIN LOOP PASS ----- **
    if (thispass == 0)
*/
    break;
  }

  free((char *) mptr2);
  return (allpasses);
}
```

```
/*****************************************************************************-
-                                                                            -
-    findbuf.c                                                               -
-                                                                            -
-    Purpose - searches the linked list of received messages for one that    -
-              meets the specifications requested.                           -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int               findbuf(cmd, name_from, mybuf)
   char            *cmd,
                   *name_from;
   struct msgbuf   *mybuf;

{
   int              msglen;

   struct msgbuf   *mptr1;

   extern struct msgbuf *msgfirst;


   for (mptr1 = msgfirst; mptr1 != NULL; mptr1 = mptr1->next) {
      if ((cmd[0] != 0x00) && (strcmp(cmd, mptr1->cmd) != 0))
         continue;
      if ((name_from[0] != 0x00) && (strcmp(name_from, mptr1->name_from) != 0))
         continue;
      break;
   }

   if (mptr1 == NULL) {
      errno = EWOULDBLOCK;
      return (-1);
   }
   sscanf(mptr1->mlen, "%d", &msglen);
   msglen += MSGBUF_MIN;
   bcopy((char *) mptr1, (char *) mybuf, msglen);

   if (msgfrmlist(mptr1) < 0) {
      printe("FINDBUF: BAD RETURN FROM MSGFRMLIST\n");
      return (-1);
   }
   return (0);
}
```

```
/*****************************************************************************
-                                                                           -
-    from.c                                                                 -
-                                                                           -
-    Purpose - called by application program to attempt to receive a message.  -
-              The desired message sender may be specifically requested.    -
-                                                                           -
-****************************************************************************/

#include            "header.h"

int             from(name, msg, msglen)
   char           *name,
                  *msg;
   int             msglen;

{
   int             rc,
                   err,
                   oldmask,
                   fromx();


   oldmask = sigblock(sigmask(SIGUSR1));
   rc = fromx(name, msg, msglen);
   err = errno;
   sigsetmask(oldmask);
   errno = err;
   return (rc);
}


static int      fromx(name, msg, msglen)
   char           *name,
                  *msg;
   int             msglen;

{
   int             minmsglen;

   char            capname[MAXNAMELEN + 1];

   struct msgbuf  *mptr2;

   extern int      ftmsock,
                   intpidme;

   extern char     intadvyet;

   extern struct msgbuf *msgfirst;

#if IDEBUG
   printf("DEBUG: FROM: SEARCH FOR NAME = %s\n", name);
#endif

   if (strnlen(name, 1) == 0) {
      errno = EINVAL;
      return (-1);
   }
   if (msglen < 0) {
      errno = EINVAL;
```

```
        return (-1);
    }
    if ((intadvyet != 'Y') || (intpidme != getpid())) {
        printe("FROM: ADV NOT SUCCESSFULLY COMPLETED\n");
        errno = EFAULT;
        return (-1);
    }
    getmsgs(-2);

    bzero(capname, sizeof(capname));
    strlcpy(capname, name, sizeof(capname) - 1);
    setcap(capname. sizeof(capname) - 1);
    if ((capname[0] == '*') && (capname[1] == 0x00))
        capname[0] = 0x00;

    if ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL) {
        perror("FROM: MALLOC");
        errno = EFAULT;
        return (-1);
    }
    bzero((char *) mptr2, MSGBUF_MAX);

    if (findbuf("TO", capname, mptr2) < 0) {
        if (errno != EWOULDBLOCK) {
            printe("FROM: BAD RETURN FROM FINDBUF\n");
            free((char *) mptr2);
            errno = EFAULT;
            return (-1);
        }
        free((char *) mptr2);
        if (ftmsock < 0)
            errno = ENOTCONN;
        else
            errno = EWOULDBLOCK;
        return (-1);
    }

#if IDEBUG
    else
        printf("DEBUG: FROM: MSG FOUND = %s %c %s %s %s %s %s %s %s\n",
                mptr2->cmd,
                mptr2->from,
                mptr2->status,
                mptr2->name_to,
                mptr2->name_from,
                mptr2->seqnum,
                mptr2->errno,
                mptr2->mlen,
                mptr2->msg);
#endif

    if (capname[0] == 0x00)
        strcpy(name, mptr2->name_from);

    sscanf(mptr2->mlen, "%d", &minmsglen);
    if (msglen < minmsglen)
        minmsglen = msglen;
    bcopy(mptr2->msg, msg, minmsglen);
    free((char *) mptr2);

    return (minmsglen);
}
```

```
/*****************************************************************************
-                                                                            -
-    ftm.c                                                                   -
-                                                                            -
-    Purpose - the main routine for the FTM (fault tolerant monitor).        -
-                                                                            -
-****************************************************************************/

#include          "header.h"

struct ftmtable *ftmptr;

int             ftmsize,
                ftmme,
                int_msgs = 0;

char            ftm_ic = 'F';

struct namtable *namfirst = NULL,
               *namlast = NULL;

struct msgbuf  *msgfirst = NULL,
               *msglast = NULL;

struct ftm_time ftmtime;


static void     report_sig(sig)
   int             sig;
{
   printe2("FTM: KILL SIGNAL RECEIVED = %d\n", sig);
   exit(-1);
}

static void     kill_sig()
{
   printe("FTM: KILL SIGNAL (SIGTERM OR SIGINT) RECEIVED\n");
   exit(-1);
}



int             main(argc, argv)
   int             argc;
   char           **argv;
{
   struct sockaddr_un server,
                   client;

   int             i,
                   s1,
                   s2,
                   len,
                   thispass,
                   num,
                   selze - -2;

   char            fname[MAXPATHLEN + 1];

   struct namtable *nptr;

   fd_set          exceptfds;
```

```
 if ((argc > 1)
     && ((strcmp(argv[1], "r") == 0) || (strcmp(argv[1], "R") == 0))) {
    FILE            *ftemp,
                    *ftemp2;
    char             thishost[MAXHOSTNAMELEN + 2];
    strcpy(thishost, "_");
    gethostname(thishost + 1, sizeof(thishost) - 1);
    strcpy(fname, SOCKPATH);
    strcat(fname, "stdxxx");
    strcat(fname, thishost);
    unlink(fname);
    if ((ftemp2 = fopen(fname, "w")) == NULL)
       exit(-1);
    strcpy(fname, SOCKPATH);
    strcat(fname, "stdout");
    strcat(fname, thishost);
    unlink(fname);
    fclose(stdout);
    if ((ftemp = fopen(fname, "w")) == NULL)
       exit(-1);
    if (fileno(ftemp) != 1)
       if (dup2(fileno(ftemp), 1) < 0)
          exit(-1);
       else
          fclose(ftemp);
    strcpy(fname, SOCKPATH);
    strcat(fname, "stderr");
    strcat(fname, thishost);
    unlink(fname);
    fclose(stderr);
    if ((ftemp = fopen(fname, "w")) == NULL)
       exit(-1);
    if (fileno(ftemp) != 2)
       if (dup2(fileno(ftemp), 2) < 0)
          exit(-1);
       else
          fclose(ftemp);
    fclose(ftemp2);
 }
 for (i = 1; i <= sizeof(int) * 8; ++i)
    if (signal(i, report_sig) == BADSIG);      /* WE REALLY DON'T CARE */
 if (signal(SIGINT, kill_sig) == BADSIG) {
    perror("FTM: SIGINT SIGNAL");
    exit(-1);
 }
 if (signal(SIGPIPE, SIG_IGN) == BADSIG) {
    perror("FTM: SIGPIPE SIGNAL");
    exit(-1);
 }
 if (signal(SIGTERM, kill_sig) == BADSIG) {
    perror("FTM: SIGTERM SIGNAL");
    exit(-1);
 }
 if (signal(SIGCONT, SIG_DFL) == BADSIG) {
    perror("FTM: SIGCONT SIGNAL");
    exit(-1);
 }
 if (signal(SIGCHLD, SIG_DFL) == BADSIG) {
    perror("FTM: SIGCHLD SIGNAL");
    exit(-1);
```

```
    }
    if (signal(SIGWINCH, SIG_DFL) == BADSIG) {
       perror("FTM: SIGWINCH SIGNAL");
       exit(-1);
    }
    if (conall() < 0) {
       printe("FTM: BAD RETURN FROM CONALL\n");
       exit(-1);
    }
    bzero(fname, sizeof(fname));
    strcpy(fname, SOCKPATH);
    strcat(fname, "ftm");
    strcat(fname, (ftmptr + ftmme)->mach);
    strcat(fname, ".sck");

    bzero((char *) &server, sizeof(server));
    server.sun_family = AF_UNIX;
    strlcpy(server.sun_path, fname, sizeof(server.sun_path));

    if ((s1 = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
       perror("FTM: SOCKET");
       exit(-1);
    }
    unlink(fname);

    if (bind(s1, (struct sockaddr *) & server, strlen(fname) + 2) < 0) {
       perror("FTM: BIND");
       printe2("FTM: FAILED ATTEMPT TO BIND UNIX SOCKET = %s\n", fname);
       exit(-1);
    }
    if (listen(s1, 5) < 0) {
       perror("CONALL: LISTEN");
       exit(-1);
    }
    if (fcntl(s1, F_SETFL, FNDELAY) < 0) {
       perror("FTM: FCNTL-1");
       exit(-1);
    }
    ftmtime.widthfds = getdtablesize();
    FD_ZERO(&ftmtime.readfds_perm);
    FD_SET(s1, &ftmtime.readfds_perm);
    for (i = 0; i < ftmsize; ++i)
       if ((ftmptr + i)->sock >= 0)
          FD_SET((ftmptr + i)->sock, &ftmtime.readfds_perm);
    ftmtime.timeval_perm.tv_sec = FTM_SLEEP_TIME / 1000000;
    ftmtime.timeval_perm.tv_usec = FTM_SLEEP_TIME % 1000000;
    ftmtime.msgtime_perm.tv_sec = MSG2_SLEEP / 1000000;
    ftmtime.msgtime_perm.tv_usec = MSG2_SLEEP % 1000000;

    printe2("FTM: %s FULLY CONNECTED - START APPLICATIONS\n",
            (ftmptr + ftmme)->mach);

  /*
   * REPEAT THE FOLLOWING LOOP FOREVER.  IT ACCEPTS NEW CONNECTIONS FROM
   * APPLICATIONS AND CHECKS FOR INCOMING MESSAGES FROM OTHER FTMs AND FROM ALL
   * CONNECTED APPLICATIONS.  WHEN IT RUNS OUT OF THINGS TO DO IT WAITS FOR SOME
   * INPUT (OR FTM_SLEEP_TIME milli-SECONDS) AND THEN TRIES AGAIN.
   */

    LOOP {

 #if FDEBUG
```

```
        {
           int             i;
           static int      yesno = FDEBUGLOOP;
           struct namtable *nptrx;
           struct msgbuf  *mptrx;

           if (yesno == 1) {
              printf("DEBUG: FTM: DUMP TABLES (0 or 1 or 2)? ");
              scanf("%d", &yesno);
              printf("\n");
           }
           if ((yesno == 1) || (yesno == 2)) {
              printf("DEBUG: FTM: SELRC = %d\n", selrc);
              if (selrc > 0)
                 for (i = 0; i < ftmtime.widthfds; ++i)
                    if (FD_ISSET(i, &ftmtime.readfds_temp))
                       printf("DEBUG: FTM: SET FD = %d\n", i);
              for (i = 0; i < ftmsize; ++i)
                 printf("DEBUG: FTM: FTM ENTRY = %s %X %d\n",
                        (ftmptr + i)->mach,
                        (ftmptr + i)->addr,
                        (ftmptr + i)->sock);
              for (nptrx = namfirst; nptrx != NULL; nptrx = nptrx->next)
                 printf("DEBUG: FTM: NAM ENTRY = %s %c %d %d %d %d %d\n",
                        nptrx->name,
                        nptrx->status,
                        nptrx->lsock,
                        nptrx->fsock,
                        nptrx->pid,
                        nptrx->bkupftm,
                        nptrx->msgs);
              for (mptrx = msgfirst; mptrx != NULL; mptrx = mptrx->next)
                 printf(
                        "DEBUG: FTM: MSG ENTRY = %s %c %s %s %s %s %s %s %s\n",
                        mptrx->cmd,
                        mptrx->from,
                        mptrx->status,
                        mptrx->name_to,
                        mptrx->name_from,
                        mptrx->seqnum,
                        mptrx->errno,
                        mptrx->mlen,
                        mptrx->msg);
              printf("DEBUG: FTM: FTMME = %d\n", ftmme);
           }
        }
#endif

        thispass = 0;
        selrc = -2;
        LOOP {
           wait3((int *) NULL, WNOHANG, (struct rusage *) NULL);
           if ((s2 = accept(s1, (struct sockaddr *) & client, &len)) < 0) {
              if (errno != EWOULDBLOCK) {
                 perror("FTM: ACCEPT");
                 exit(-1);
              } else
                 break;
           } else if (fcntl(s2, F_SETFL, FNDELAY) < 0) {
              perror("FTM: FCNTL-2");
              exit(-1);
           } else if (setsockopt(s2, SOL_SOCKET, SO_KEEPALIVE, (char *) 0, 0)
```

```
                < 0) {
            perror("FTM: SETSOCKOPT-1");
            exit(-1);
         } else if (setsockopt(s2, SOL_SOCKET, SO_DONTLINGER, (char *) 0, 0)
                < 0) {
            perror("FTM: SETSOCKOPT-2");
            exit(-1);
         } else if (namtolist("", s2, -1, 0, -1) < 0) {
            printe("FTM: BAD RETURN FROM NAMTOLIST\n");
            exit(-1);
         } else {

#if FDEBUG
            printf("DEBUG: FTM: ACCEPT ON SOCKET = %d\n", s2);
#endif

            FD_SET(s2, &ftmtime.readfds_perm);
            ++thispass;
         }
      }

#if FDEBUG
      if (thispass > 0)
         printf("DEBUG: FTM: GOT %d ACCEPTS\n", thispass);
#endif

      if ((num = ffget()) < 0) {
         if (errno == EWOULDBLOCK);
         else if (errno == ENOTCONN) {
            ;
         } else {
            printe("FTM: BAD RETURN FROM FFGET\n");
            exit(-1);
         }
      } else
         thispass += num;

#if FDEBUG
      if (num > 0)
         printf("DEBUG: FTM: GOT %d ACTIONS FROM FFGET\n", num);
#endif

      if ((num = ifget()) < 0)
         if (errno == EWOULDBLOCK);      /* OK - NO MORE INT MSGS */
         else if (errno == ENOTCONN);    /* TAKEN CARE OF IN IFGET */
         else {
            printe("FTM: BAD RETURN FROM IFGET\n");
            exit(-1);
         }
      else
         thispass += num;

#if FDEBUG
      if (num > 0)
         printf("DEBUG: FTM: GOT %d ACTIONS FROM IFGET\n", num);
#endif

      if ((num = msghndl()) < 0) {
         printe("FTM: BAD RETURN FROM MSGHNDL\n");
         exit(-1);
      } else
         thispass += num;
```

```
#if FDEBUG
      if (num > 0)
         printf("DEBUG: FTM: GOT %d ACTIONS FROM MSGHNDL\n", num);
#endif

      if ((int_msgs > MSG_LIMIT) || (thispass == 0)) {
         for (nptr = namfirst; nptr != NULL; nptr = nptr->next)
            if (nptr->msgs > 0) {

#if FDEBUG
               printf("DEBUG: FTM: SEND SIGUSR1 TO PID = %d\n",
                      nptr->pid);
#endif

               kill(nptr->pid, SIGUSR1);
               nptr->msgs = 0;
            }
         int_msgs = 0;
      }
      if (thispass == 0) {
         bcopy((char *) &ftmtime.readfds_perm, (char *) &ftmtime.readfds_temp,
               sizeof(struct fd_set));
         bcopy((char *) &ftmtime.readfds_perm, (char *) &exceptfds,
               sizeof(struct fd_set));
         bcopy((char *) &ftmtime.timeval_perm, (char *) &ftmtime.timeval_temp,
               sizeof(struct timeval));
         if ((selrc = select(ftmtime.widthfds, &ftmtime.readfds_temp,
                             (fd_set *) NULL, &exceptfds,
                             &ftmtime.timeval_temp)) < 0) {
            perror("FTM: SELECT");
            printe("FTM: TERMINATED BECAUSE OF ERROR ON SELECT CALL\n");
            exit(-1);
         }
      }
   }
}
```

```
/*****************************************************************************
-                                                                            -
-    ftmclose.c                                                              -
-                                                                            -
-    Purpose - called by application program to invalidate its advertised    -
-              name. If the program terminates before it closes its          -
-              advertised name, the fault tolerant system treats the         -
-              termination as abnormal.                                      -
-                                                                            -
-****************************************************************************/

#include        "header.h"

int             ftmclose()
{
   int             rc,
                   err,
                   oldmask,
                   ftmclosex();


   oldmask = sigblock(sigmask(SIGUSR1));
   rc = ftmclosex();
   err = errno;
   sigsetmask(oldmask);
   errno = err;
   return (rc);
}



static int      ftmclosex()
{
   struct msgbuf  *mptr,
                  *mptr2;

   extern char     intadvyet,
                   intmyname[MAXNAMELEN + 1];

   extern int      ftmsock,
                   intseqnum;

   extern struct ftm_time ftmtime;


   if (intadvyet != 'Y') {
      printe("FTMCLOSE: ADV NOT SUCCESSFULLY COMPLETED\n");
      errno = EFAULT;
      return (-1);
   }
   if (ftmsock < 0) {
      errno = ENOTCONN;
      return (-1);
   }
   if (((mptr = (struct msgbuf *) malloc(MSGBUF_MIN)) == NULL)
       || ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MIN)) == NULL)) {
      perror("FTMCLOSE: MALLOC");
      if (mptr != NULL)
         free((char *) mptr);
      errno = EFAULT;
      return (-1);
   }
```

```
   bzero((char *) mptr, MSGBUF_MIN);
   bzero((char *) mptr2, MSGBUF_MIN);

   strcpy(mptr->cmd, "CLOSE");
   mptr->from = 'I';
   sprintf(mptr->status, "%d", -1);
   strcpy(mptr->name_from, intmyname);
   sprintf(mptr->seqnum, "%d", intseqnum);
   ++intseqnum;
   strcpy(mptr->errno, "0");
   strcpy(mptr->mien, "0");

   if (send_msg(ftmsock, mptr->cmd, MSGBUF_MIN - MSGBUF_HDR) < 0)
      if ((errno != EPIPE) && (errno != ENOTCONN)) {
         perror("FTMCLOSE: SEND_MSG");
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      }

#if IDEBUG
   printf("DEBUG: FTMCLOSE: CLOSE MESSAGE SENT TO LOCAL FTM = %d\n", ftmsock);
   printf("DEBUG: FTMCLOSE: NOW WAITING FOR CLOSOK\n");
#endif

 /*
  * LOOP UNTIL THE CLOSOK MESSAGE IS RECEIVED
  */

   LOOP {
      getmsgs(-3);

      if (findbuf("CLOSOK", "", mptr2) < 0) {
         if (errno != EWOULDBLOCK) {
            printe("FTMCLOSE: BAD RETURN FROM FINDBUF-1\n");
            free((char *) mptr);
            free((char *) mptr2);
            errno = EFAULT;
            return (-1);
         } else;
      } else if (strcmp(mptr2->name_to, intmyname) != 0) {
         printe2("FTMCLOSE: NAME IN CLOSOK IS %s\n", mptr2->name_to);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else if (strcmp(mptr->seqnum, mptr2->seqnum) != 0) {
         printe3("FTMCLOSE: CLOSOK SEQNUM MISMATCH %s %s\n", mptr->seqnum,
                 mptr2->seqnum);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else {

#if IDEBUG
         printf("DEBUG: FTMCLOSE: CLOSOK MESSAGE RECEIVED\n");
#endif

         free((char *) mptr);
         free((char *) mptr2);
```

```
            FD_CLR(ftmsock, &ftmtime.readfds_perm);
            close(ftmsock);
            ftmsock = -2;
            intadvyet = 'R';
            return (0);
         }
         if (ftmsock < 0) {
            errno = EWOULDBLOCK;
            return (-1);
         }
         bcopy((char *) &ftmtime.readfds_perm, (char *) &ftmtime.readfds_temp,
               sizeof(struct fd_set));
         bcopy((char *) &ftmtime.timeval_perm, (char *) &ftmtime.timeval_temp,
               sizeof(struct timeval));
         select(ftmtime.widthfds, &ftmtime.readfds_temp, NULL, NULL,
                &ftmtime.timeval_temp);
      }
   }
```

```
/*****************************************************************************-
-                                                                            -
-    ftmisdown.c                                                             -
-                                                                            -
-    Purpose - called by an ftm when it discovers than another ftm is no     -
-              longer accessible.                                            -
-                                                                            -
-****************************************************************************/

#include            "header.h"

int             ftmisdown(fptr)
   struct ftmtable *fptr;

{
   int             orig_sock;

   struct namtable *nptr;

   struct msgbuf  *mptr;

   extern struct namtable *namfirst;

   extern struct ftm_time ftmtime;


   orig_sock = -3 - fptr->sock;
   close(orig_sock);
   FD_CLR(orig_sock, &ftmtime.readfds_perm);

   if ((mptr = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL) {
      perror("FTMISDOWN: MALLOC");
      errno = EFAULT;
```

```
        return (-1);
    }
    for (nptr = namfirst; nptr != NULL; nptr = nptr->next) {
        if (nptr->fsock != orig_sock)
            break;
        bzero((char *) mptr, MSGBUF_MAX);
        strcpy(mptr->cmd, "KILLN");
        mptr->from = 'I';
        strcpy(mptr->status, "-1");
        strcpy(mptr->name_from, nptr->name);
        strcpy(mptr->seqnum, "0");
        sprintf(mptr->errno, "%d", -2);
        sprintf(mptr->msg, "%X", nptr);
        sprintf(mptr->mlen, "%d", strlen(mptr->msg));

#if FDEBUG
        printf("DEBUG: FTMISDOWN: MSG QUEUED: %s %c %s %s %s %s %s %s %s\n",
                mptr->cmd,
                mptr->from,
                mptr->status,
                mptr->name_to,
                mptr->name_from,
                mptr->seqnum,
                mptr->errno,
                mptr->mlen,
                mptr->msg);
#endif

        if (msgtolist(mptr) < 0) {
            printe("FTMISDOWN: BAD RETURN FROM MSGTOLIST\n");
            free((char *) mptr);
            errno = EFAULT;
            return (-1);
        }
    }

    free((char *) mptr);
    return (0);
}
```

```
/*****************************************************************************
-                                                                            -
-    ftmkill.c                                                               -
-                                                                            -
-    Purpose - invoked to kill all running ftms on auvc1, auvc2,..., auvc16. -
-              Uses shell script "rkill".                                    -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int               main()
{
   int               rc;

   char              line[sizeof("rkill -TERM ") - 1
                          +                MAXPATHLEN       /* FTM PATH */
                          +                1];


   strcpy(line, "rkill -TERM ");
   strcat(line, FTMPATH);
   strcat(line, FTMPROG);
   if ((rc = system(line)) < 0)
      perror("FTMKILL: SYSTEM(LINE)");
   exit(rc);
}
```

```
/*****************************************************************************
-                                                                            -
-    ftmstart.c                                                              -
-                                                                            -
-    Purpose - invoked to initiate the fault tolerant system. Causes daemons -
-              to start FTMs on all systems as defined in the ftms.lis file. -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int               main()
{
   int               rc,
                     systems = 0;

   char              line[sizeof("rsh ") - 1
                          +                MAXHOSTNAMELEN
                          +                sizeof(" -n ") - 1
                          +                MAXPATHLEN       /* DAEMON PATH */
                          +                1
                          +                MAXPATHLEN       /* FTM PATH */
                          +                sizeof(" r") - 1
                          +                1];
```

```
char            ftmsfile[MAXPATHLEN + 1],
                thishost[MAXHOSTNAMELEN + 1],
                hostme[MAXHOSTNAMELEN + 1];

struct hostent *hp;

FILE           *filein;


if (gethostname(hostme, sizeof(hostme)) < 0) {
   perror("FTMSTART: GETHOSTNAME");
   exit(-1);
}
hostme[sizeof(hostme) - 1] = 0x00;
if ((hp = gethostbyname(hostme)) == NULL) {
   perror("FTMSTART: GETHOSTBYNAME-1");
   exit(-1);
} else if (hp->h_length != 4) {
   printe2("FTMSTART: ILLEGAL NET ADDR LENGTH-1 = %d\n", hp->h_length);
   exit(-1);
} else if (hp->h_addrtype != AF_INET) {
   printe2("FTMSTART: ILLEGAL ADDRTYPE-1 = %d\n", hp->h_addrtype);
   exit(-1);
}
strcpy(ftmsfile, FTMPATH);
strcat(ftmsfile, FTMFILE);
if ((filein = fopen(ftmsfile, "r")) == NULL) {
   perror("FTMSTART: INPUT FILE FOPEN ERROR");
   exit(-1);
}
while (fgets(thishost, sizeof(thishost), filein) != NULL) {
   if (thishost[0] == '\n')
      continue;
   if (thishost[0] == '*') {
      while (thishost[strlen(thishost) - 1] != '\n')
         if (fgets(thishost, sizeof(thishost), filein) == NULL) {
            printe2("FTMSTART: ERROR DURING READ OF %s\n", FTMFILE);
            exit(-1);
         }
      continue;
   }
   nline(thishost, sizeof(thishost));
   strcpy(line, "rsh ");
   strcat(line, thishost);
   strcat(line, " -n ");
   strcat(line, FTMPATH);
   strcat(line, DAEMONPROG);
   strcat(line, " ");
   strcat(line, FTMPATH);
   strcat(line, FTMPROG);
   strcat(line, " r");
   if ((rc = fork()) < 0) {
      perror("FTMSTART: FORK");
      exit(-1);
   } else if (rc == 0) {
      printe2("FTMSTART: ABOUT TO START FTM ON %s\n", thishost);
      if ((rc = system(line)) != 0)
         perror("FTMSTART: SYSTEM(LINE)");
      exit(rc);
   }
   ++systems;
```

```
    }

    fclose(filein);

    if (systems == 0) {
       printe("FTMSTART: EMPTY FTMS.LIS FILE`.");
       exit(-1);
    }
    exit(0);
}
```

```
/*****************************************************************************
-                                                                            -
-    ftmstatus.c                                                             -
-                                                                            -
-    Purpose - called by application program to retrieve global status      -
-              information from the local FTM.                               -
-                                                                            -
-****************************************************************************/

#include            "header.h"

int             ftmstatus(pnum, sunstat, mirror)
   int             pnum,
                  *sunstat,
                  *mirror;

/*
sunstat: BIT 0  => auvsun0
         BIT 1  => auvc1
         BIT 2  => auvc2
         BIT 3  => auvc3
         BIT 4  => auvc4
         BIT 5  => auvc5
         BIT 6  => auvc6
         BIT 7  => auvc7
         BIT 8  => auvc8
         BIT 9  => auvc9
         BIT 10 => auvc10
         BIT 11 => auvc11
         BIT 12 => auvc12
         BIT 13 => auvc13
         BIT 14 => auvc14
         BIT 15 => auvc15
         BIT 16 => auvc16
         BIT 17 => auvsun1

mirror == 0 => DRIVE 0 DOWN   *** MEANINGLESS UNTIL DISK MIRRORING INSTALLED
       == 1 >= DRIVE 1 DOWN
       == 2 >= BOTH DRIVES UP
*/
```

```
{
   int             rc,
                   err,
                   oldmask,
                   ftmstatusx();


   oldmask = sigblock(sigmask(SIGUSR1));
   rc = ftmstatusx(pnum, sunstat, mirror);
   err = errno;
   sigsetmask(oldmask);
   errno = err;
   return (rc);
}


static int      ftmstatusx(pnum, sunstat, mirror)
   int             pnum,
                  *sunstat,
                  *mirror;

{
   int             errx,
                   sunstatx,
                   mirrorx;

   struct msgbuf  *mptr,
                  *mptr2;

   extern char     intadvyet,
                   intmyname[MAXNAMELEN + 1];

   extern int      ftmsock,
                   intseqnum,
                   intpidme;

   extern struct ftm_time ftmtime;


   if ((intadvyet != 'Y') || (intpidme != getpid())) {
      printe("FTMSTATUS: ADV NOT YET COMPLETED SUCCESSFULLY\n");
      errno = EFAULT;
      return (-1);
   }
   if (ftmsock < 0) {
      errno = ENOTCONN;
      return (-1);
   }
   if (((mptr = (struct msgbuf *) malloc(MSGBUF_MIN)) == NULL)
       || ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL)) {
      perror("FTMSTATUS: MALLOC");
      if (mptr != NULL)
         free((char *) mptr);
      errno = EFAULT;
      return (-1);
   }
   bzero((char *) mptr, MSGBUF_MIN);
   bzero((char *) mptr2, MSGBUF_MAX);

   strcpy(mptr->cmd, "STATUS");
   mptr->from = 'I';
   sprintf(mptr->status, "%d", -1);
```

```
   strcpy(mptr->name_from, intmyname);
   sprintf(mptr->seqnum, "%d", intseqnum);
   ++intseqnum;
   strcpy(mptr->errno, "0");
   strcpy(mptr->mlen, "0");

   if (send_msg(ftmsock, mptr->cmd, MSGBUF_MIN - MSGBUF_HDR) < 0) {
      errx = errno;
      perror("FTMSTATUS: SEND_MSG");
      free((char *) mptr);
      free((char *) mptr2);
      if (errx == ENOTCONN)
         errno = ENOTCONN;
      else
         errno = EFAULT;
      return (-1);
   }

#if IDEBUG
   printf("DEBUG: FTMSTATUS: STATUS MESSAGE SENT TO LOCAL FTM = %d\n",
          ftmsock);
   printf("DEBUG: FTMSTATUS: NOW WAITING FOR STATOK\n");
#endif

 /*
  * LOOP UNTIL THE STATOK MESSAGE IS RECEIVED
  */

   LOOP {
      getmsgs(-4);

      if (findbuf("STATOK", "", mptr2) < 0) {
         if (errno != EWOULDBLOCK) {
            printe("FTMSTATUS: BAD RETURN FROM FINDBUF-1\n");
            free((char *) mptr);
            free((char *) mptr2);
            errno = EFAULT;
            return (-1);
         } else;
      } else if (strcmp(mptr2->name_to, intmyname) != 0) {
         printe2("FTMSTATUS: NAME IN STATOK IS %s\n", mptr2->name_to);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else if (strcmp(mptr->seqnum, mptr2->seqnum) != 0) {
         printe3("FTMSTATUS: STATOK SEQNUM MISMATCH %s %s\n",
                 mptr->seqnum, mptr2->seqnum);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else {

#if IDEBUG
         printf("DEBUG: FTMSTATUS: STATOK MESSAGE RECEIVED\n");
#endif

         sscanf(mptr2->msg, "%X %X", &sunstatx, &mirrorx);
         if (pnum >= 1)
            *sunstat = sunstatx;
         if (pnum >= 2)
```

```
            *mirror = mirrorx;
         free((char *) mptr);
         free((char *) mptr2);
         return (0);
      }

      if (ftmsock < 0) {
         free((char *) mptr);
         free((char *) mptr2);
         errno = ENOTCONN;
         return (-1);
      }
      bcopy((char *) &ftmtime.readfds_perm, (char *) &ftmtime.readfds_temp,
            sizeof(struct fd_set));
      bcopy((char *) &ftmtime.timeval_perm, (char *) &ftmtime.timeval_temp,
            sizeof(struct timeval));
      select(ftmtime.widthfds, &ftmtime.readfds_temp, NULL, NULL,
             &ftmtime.timeval_temp);
   }
}
```

```
/*****************************************************************************-
-                                                                            -
-    ftmwhere.c                                                              -
-                                                                            -
-    Purpose - called by application program to request information about    -
-              the current and backup physical machines for any program.     -
-                                                                            -
-****************************************************************************/

#include      "header.h"

int             ftmwhere(name, mach, bkup)
   char           *name,
                  *mach,
                  *bkup;

{
   int             rc,
                   err,
                   oldmask,
                   ftmwherex();


   oldmask = sigblock(sigmask(SIGUSR1));
   rc = ftmwherex(name, mach, bkup);
   err = errno;
   sigsetmask(oldmask);
   errno = err;
   return (rc);
}
```

```
static int      ftmwherex(name, mach, bkup)
   char           *name,
                  *mach,
                  *bkup;

{
   char            capname[MAXNAMELEN + 1];

   struct msgbuf  *mptr,
                  *mptr2;

   int             errx;

   extern char     intadvyet,
                   intmyname[MAXNAMELEN + 1];

   extern int      ftmsock,
                   intseqnum,
                   intpidme;

   extern struct ftm_time ftmtime;


   if ((intadvyet != 'Y') || (intpidme != getpid())) {
      printe("FTMWHERE: ADV NOT YET COMPLETED SUCCESSFULLY\n");
      errno = EFAULT;
      return (-1);
   }
   if (ftmsock < 0) {
      errno = ENOTCONN;
      return (-1);
   }
   bzero(capname, sizeof(capname));
   strlcpy(capname, name, sizeof(capname) - 1);
   setcap(capname, sizeof(capname) - 1);
   if ((capname[0] == 0x00) || (strcmp(capname, "*") == 0)) {
      errno = EINVAL;
      return (-1);
   }
   if (((mptr = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL)
       || ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL)) {
      perror("FTMWHERE: MALLOC");
      if (mptr != NULL)
         free((char *) mptr);
      errno = EFAULT;
      return (-1);
   }
   bzero((char *) mptr, MSGBUF_MAX);
   bzero((char *) mptr2, MSGBUF_MAX);

   strcpy(mptr->cmd, "WHERE");
   mptr->from = 'I';
   sprintf(mptr->status, "%d", -1);
   strcpy(mptr->name_from, intmyname);
   sprintf(mptr->seqnum, "%d", intseqnum);
   ++intseqnum;
   strcpy(mptr->errno, "0");
   strcpy(mptr->msg, capname);
   sprintf(mptr->mlen, "%d", strlen(mptr->msg));
```

```
   if (send_msg(ftmsock, mptr->cmd,
                MSGBUF_MIN - MSGBUF_HDR + strlen(mptr->msg)) < 0) {
      errx = errno;
      perror("FTMWHERE: SEND_MSG");
      free((char *) mptr);
      free((char *) mptr2);
      if (errx == ENOTCONN)
         errno = ENOTCONN;
      else
         errno = EFAULT;
      return (-1);
   }

#if IDEBUG
   printf("DEBUG: FTMWHERE: WHERE MESSAGE SENT TO LOCAL FTM = %d\n", ftmsock);
   printf("DEBUG: FTMWHERE: NOW WAITING FOR WHEROK\n");
#endif

 /*
  * LOOP UNTIL THE WHEROK MESSAGE IS RECEIVED
  */

   LOOP {
      getmsgs(-5);

      if (findbuf("WHEROK", "", mptr2) < 0) {
         if (errno != EWOULDBLOCK) {
            printe("FTMWHERE: BAD RETURN FROM FINDBUF-1\n");
            free((char *) mptr);
            free((char *) mptr2);
            errno = EFAULT;
            return (-1);
         } else;
      } else if (strcmp(mptr2->name_to, intmyname) != 0) {
         printe2("FTMWHERE: NAME IN WHEROK IS %s\n", mptr2->name_to);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else if (strcmp(mptr->seqnum, mptr2->seqnum) != 0) {
         printe3("FTMWHERE: WHEROK SEQNUM MISMATCH %s %s\n", mptr->seqnum,
                 mptr2->seqnum);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else {

#if IDEBUG
         printf("DEBUG: FTMWHERE: WHEROK MESSAGE RECEIVED\n");
#endif

         strcpy(mach, mptr2->msg);
         strcpy(bkup, (mptr2->msg) + MAXHOSTNAMELEN + 1);
         free((char *) mptr);
         free((char *) mptr2);
         return (0);
      }

      if (ftmsock < 0) {
         errno = ENOTCONN;
         return (-1);
```

```
        }
        bcopy((char *) &ftmtime.readfds_perm, (char *) &ftmtime.readfds_temp,
              sizeof(struct fd_set));
        bcopy((char *) &ftmtime.timeval_perm, (char *) &ftmtime.timeval_temp,
              sizeof(struct timeval));
        select(ftmtime.widthfds, &ftmtime.readfds_temp, NULL, NULL,
               &ftmtime.timeval_temp);
    }
}
```

```
/*****************************************************************************
-                                                                           -
-    getmsgs.c                                                              -
-                                                                           -
-    Purpose - routine to retrieve message(s) by the intercept code and     -
-              insert them into a linked list. This routine is sometimes    -
-              called synchronously and is sometimes called as an interrupt -
-              handler.                                                     -
-                                                                           -
-****************************************************************************/

#include    "header.h"

void            getmsgs(sig)
   int             sig;

{
   int             len,
                   err;

   char            tempname[MAXNAMELEN + 1];

   struct msgbuf  *mptr2;

   extern int      ftmsock;

   extern struct msgbuf *msgfirst;

   extern struct ftm_time ftmtime;


   err = errno;


   if (ftmsock < 0) {
      errno = err;
      return;
   }
   if ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL) {
      perror("GETMSGS: MALLOC");
      printe("GETMSGS: CRITICAL HANDLER ERROR - PROCESS ABORTED\n");
      exit(-1);
```

```
    }
  /*
   * LOOP UNTIL NO MORE MESSAGES TO RECEIVE
   */

    LOOP {
       if ((len = recv_msg(ftmsock, mptr2->cmd, MSGBUF_MAX - MSGBUF_HDR)) < 0) {
          if (errno == EWOULDBLOCK)
             break;
          else if (errno == ENOTCONN) {

#if IDEBUG
             printf("DEBUG: GETMSGS: SOCKET CLOSED TO LOCAL FTM = %d\n",
                    ftmsock);
#endif

             free((char *) mptr2);
             FD_CLR(ftmsock, &ftmtime.readfds_perm);
             close(ftmsock);
             ftmsock = -2;
             errno = err;
             return;
          } else {
             perror("GETMSGS: RECV_MSG");
             printe("GETMSGS: CRITICAL HANDLER ERROR - PROCESS ABORTED\n");
             free((char *) mptr2);
             exit(-1);
          }
       }

#if IDEBUG
       printf("DEBUG: GETMSGS: MSG FOUND = %s %c %s %s %s %s %s %s %s\n",
              mptr2->cmd,
              mptr2->from,
              mptr2->status,
              mptr2->name_to,
              mptr2->name_from,
              mptr2->seqnum,
              mptr2->errno,
              mptr2->mlen,
              mptr2->msg);
#endif

       if (msgtolist(mptr2) < 0) {
          printe("GETMSGS: BAD RETURN FROM MSGTOLIST\n");
          printe("GETMSGS: CRITICAL HANDLER ERROR - PROCESS ABORTED\n");
          free((char *) mptr2);
          exit(-1);
       }
       if (strcmp(mptr2->cmd, "TO") == 0) {
          strcpy(mptr2->cmd, "TOOK");
          mptr2->from = 'I';
          strcpy(tempname, mptr2->name_to);
          strcpy(mptr2->name_to, mptr2->name_from);
          strcpy(mptr2->name_from, tempname);

#if IDEBUG
          printf("DEBUG: GETMSGS: ABOUT TO SEND A TOOK TO FTM\n");
#endif

          if (send_msg(ftmsock, mptr2->cmd, len) < 0) {
             perror("GETMSGS: SEND_MSG");
```

```
            printe("GETMSGS: CRITICAL HANDLER ERROR - PROCESS ABORTED\n");
            free((char *) mptr2);
            exit(-1);
        }
    }
  }

  free((char *) mptr2);
  errno = err;
  return;
}
```

```
/*************************************************************************-
-                                                                        -
-    hndladvf.c                                                          -
-                                                                        -
-    Purpose - handler routine to operate on messages from other FTMs which -
-              attempt to globally advertise a name.                     -
-                                                                        -
-*************************************************************************/

#include          "header.h"

int             hndladvf(mptr, mptr2, delmptr)
   struct msgbuf  *mptr,
                  *mptr2;
   char           *delmptr;

/*
 * ------------------------------------------------------------------
 *
 * handle ADV messages from FTM
 *
 * ------------------------------------------------------------------
 */

{
   int             mptrlen,
                   mptrlen2,
                   i,
                   lsock,
                   fsock,
                   pid,
                   bkupftm;

   extern struct ftmtable *ftmptr;

   extern int      ftmsize;


   *delmptr = 'N';
   sscanf(mptr->mlen, "%d", &mptrlen);
```

```
   mptrlen += MSGBUF_MIN;

   sscanf(mptr->msg, "%d %d %d %d", &pid, &lsock, &bkupftm, &fsock);
   bzero(mptr2->cmd, mptrlen - MSGBUF_HDR);
   strcpy(mptr2->cmd, "ADVOK");
   mptr2->from = 'F';
   sprintf(mptr2->status, "%d", -1);
   strcpy(mptr2->name_to, mptr->name_from);
   strcpy(mptr2->seqnum, mptr->seqnum);
   strcpy(mptr2->errno, "0");
   sprintf(mptr2->msg, "%d %d %d", pid, lsock, bkupftm);
   mptrlen2 = MSGBUF_MIN + strlen(mptr2->msg);
   sprintf(mptr2->mlen, "%d", strlen(mptr2->msg));
   if ((namtolist(mptr->name_from, lsock, fsock, pid, bkupftm)) < 0)
      if (errno == EWOULDBLOCK) {
         strcpy(mptr2->cmd, "ADVNO");
         sprintf(mptr2->errno, "%d", EWOULDBLOCK);
      } else {
         printe("HNDLADVF: BAD RETURN FROM NAMTOLIST\n");
         return (-1);
      }

#if FDEBUG
   printf("DEBUG: HNDLADVF: ADVOK MSG TO SEND = %s %c %s %s %s %s %s %s %s\n",
          mptr2->cmd,
          mptr2->from,
          mptr2->status,
          mptr2->name_to,
          mptr2->name_from,
          mptr2->seqnum,
          mptr2->errno,
          mptr2->mlen,
          mptr2->msg);
#endif

   if (send_msg(fsock, mptr2->cmd, mptrlen2 - MSGBUF_HDR) < 0)
      if (errno != ENOTCONN) {
         perror("HNDLADVF: SEND_MSG");
         return (-1);
      } else {
         for (i = 0; i < ftmsize; ++i)
            if ((ftmptr + i)->sock == fsock) {
               (ftmptr + i)->sock = -3 - fsock;
               break;
            }
      }
   *delmptr = 'Y';
   return (1);
}
```

```
/*****************************************************************************-
-                                                                            -
-    hndladvi.c                                                              -
-                                                                            -
-    Purpose - handler routine to operate on messages from connected         -
-              applications advertising their name.                          -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int             hndladvi(mptr, mptr2, delmptr)
   struct msgbuf  *mptr,
                  *mptr2;
   char           *delmptr;

/*
 * ------------------------------------------------------------------
 *
 * handle ADV messages from INT
 *
 * ------------------------------------------------------------------
 */

{
   int             i,
                   s,
                   mptrlen,
                   mptrlen2,
                   lsock,
                   pid,
                   num,
                   temp1,
                   temp2,
                   temp3,
                   temp4,
                   temp5,
                   bkupftm;

   struct namtable *nptr;

   extern struct ftmtable *ftmptr;
   extern int      ftmsize;

   extern struct ftm_time ftmtime;


   *delmptr = 'N';
   sscanf(mptr->mlen, "%d", &mptrlen);
   mptrlen += MSGBUF_MIN;

   sscanf(mptr->msg, "%d %d %d", &pid, &lsock  &bkupftm);
   bcopy(mptr->cmd, mptr2->cmd, mptrlen - MSGBUF_HDR);
   mptrlen2 = mptrlen;
   mptr2->from = 'F';
   if ((namtolist(mptr->name_from, lsock, -1, pid, bkupftm)) < 0)
      if (errno != EWOULDBLOCK) {
         printe("HNDLADVI: BAD RETURN FROM NAMTOLIST\n");
         return (-1);
      } else {
         temp1 = lsock;
         if (namlookup("", &temp1, &temp2, &temp3, &temp4, &temp5, &nptr) < 0) {
```

```
            printe("HNDLADVI: BAD RETURN FROM NAMLOOKUP\n");
            return (-1);
         } else if (namfrmlist(nptr) < 0) {
            printe("HNDLADVI: BAD RETURN FROM NAMFRMLIST\n");
            return (-1);
         }
         strcpy(mptr2->cmd, "ADVNO");
         strcpy(mptr2->name_to, mptr2->name_from);
         bzero(mptr2->name_from, sizeof(mptr2->name_from));
         sprintf(mptr2->errno, "%d", EWOULDBLOCK);
         bzero(mptr2->msg, strlen(mptr2->msg));
         sprintf(mptr2->msg, "%d", pid);
         bzero(mptr2->mlen, sizeof(mptr2->mlen));
         sprintf(mptr2->mlen, "%d", strlen(mptr2->msg));
         mptrlen2 = MSGBUF_MIN + strlen(mptr2->msg);

#if FDEBUG
         printf
            ("DEBUG: HNDLADVI: ADVNO MESSAGE RETURNED TO LSOCK = %d\n",
             lsock);
         printf("DEBUG: HNDLADVI: MSG SENT = %s %c %s %s %s %s %s %s %s\n",
                mptr2->cmd,
                mptr2->from,
                mptr2->status,
                mptr2->name_to,
                mptr2->name_from,
                mptr2->seqnum,
                mptr2->errno,
                mptr2->mlen,
                mptr2->msg);
#endif

         if (send_msg(lsock, mptr2->cmd, mptrlen2 - MSGBUF_HDR) < 0) {
            perror("HNDLADVI: SEND_MSG-1");
            return (-1);
         } else {
            close(lsock);
            kill(pid, SIGUSR1);
            FD_CLR(lsock, &ftmtime.readfds_perm);
            *delmptr = 'Y';
            return (1);
         }
      }

   num = 0;
   for (i = 0; i < ftmsize; ++i)
      if ((s = (ftmptr + i)->sock) >= 0)
         if (send_msg(s, mptr2->cmd, mptrlen2 - MSGBUF_HDR) < 0) {
            if (errno != ENOTCONN) {
               perror("HNDLADVI: SEND_MSG-2");
               return (-1);
            } else {
               (ftmptr + i)->sock = -3 - s;
            }
         } else {
            ++num;

#if FDEBUG
            printf
               ("DEBUG: HNDLADVI: ADV MESSAGE %d SENT TO FSOCK = %d\n",
                num, s);
            printf("DEBUG: HNDLADVI: MSG SENT = %s %c %s %s %s %s %s %s %s\n",
```

```
                    mptr2->cmd,
                    mptr2->from,
                    mptr2->status,
                    mptr2->name_to,
                    mptr2->name_from,
                    mptr2->seqnum,
                    mptr2->errno,
                    mptr2->mlen,
                    mptr2->msg);
#endif

        }
   if (num == 0) {
      bzero(mptr2->cmd, sizeof(mptr2->cmd));
      strcpy(mptr2->cmd, "ADVOK");
      mptr2->from = 'F';
      bzero(mptr2->status, sizeof(mptr2->status));
      sprintf(mptr2->status, "%d", -1);
      strcpy(mptr2->name_to, mptr2->name_from);
      bzero(mptr2->name_from, sizeof(mptr2->name_from));
      if (msgtolist(mptr2) < 0) {
         printe("MSGHNDL: ADV(I) - BAD RETURN FROM MSGTOLIST\n");
         return (-1);
      }
      num = 1;
   }
   bzero(mptr->status, sizeof(mptr->status));
   sprintf(mptr->status, "%d", num);
   return (num + 1);
}
```

```
/*****************************************************************************-
-                                                                            -
-    hndladvok.c                                                             -
-                                                                            -
-    Purpose - handler routine to operate on messages from other FTMs which  -
-              reply in the positive or negative to an attempt to globally   -
-              advertise a name. This is the response to the first phase of  -
-              the 2-phase commit protocol used for name advertising.        -
-                                                                            -
-*****************************************************************************/

#include          "header.h"

int             hndladvok(mptr, mptr2, delmptr)
   struct msgbuf  *mptr,
                  *mptr2;
   char           *delmptr;

/*
 * ---------------------------------------------------------------------
 *
```

```
 * handle ADVOK and ADVNO messages
 *
 * ------------------------------------------------------------------
 */

{
   int             i,
                   fsock,
                   lsock,
                   pid,
                   msgs,
                   num,
                   status,
                   bkupftm,
                   errx;

   struct namtable *nptr;

   extern struct ftmtable *ftmptr;
   extern int      ftmsize;

   extern struct ftm_time ftmtime;


   *delmptr = 'N';
   num = 0;
   if (findbuf("ADV", mptr->name_to, mptr2) < 0) {
      *delmptr = 'Y';
      return (1);
   }
   if (strcmp(mptr2->seqnum, mptr->seqnum) != 0) {
      printe3("HNDLADVOK: SEQNUM MISMATCH %s %s\n", mptr->seqnum,
              mptr2->seqnum);
      return (-1);
   }
   if (strcmp(mptr->cmd, "ADVNO") == 0)
      sprintf(mptr2->errno, "%d", EWOULDBLOCK);
   sscanf(mptr2->status, "%d", &status);
   --status;
   if (status > 0) {
      bzero(mptr2->status, sizeof(mptr2->status));
      sprintf(mptr2->status, "%d", status);
      *delmptr = 'Y';
      if (msgtolist(mptr2) < 0) {
         printe("HNDLADVOK: BAD RETURN FROM MSGTOLIST-1\n");
         return (-1);
      } else
         return (1);
   }
   if (namlookup(mptr2->name_from, &lsock, &fsock, &pid, &bkupftm,
                 &msgs, &nptr) < 0) {
      printe2("HNDLADVOK: BAD RETURN FROM NAMLOOKUP FOR NAME = %s\n",
              mptr2->name_from);
      *delmptr = 'Y';
      return (-1);
   }
   if (fsock != -1) {
      printe2("HNDLADVOK: FSOCK IS NOT ME = %d\n", fsock);
      *delmptr = 'Y';
      return (-1);
   }
   strcpy(mptr2->cmd, mptr->cmd);
```

```
   sscanf(mptr2->errno, "%d", &errx);
   if (errx == 0x00)
      nptr->status = 'Y';
   else {
      if (namfrmlist(nptr) < 0) {
         printe("HNDLADVOK: BAD RETURN FROM NAMFRMLIST-1\n");
         return (-1);
      } else
         strcpy(mptr2->cmd, "ADVNO");
   }
   mptr2->from = 'F';
   sprintf(mptr2->status, "%d", -1);
   strcpy(mptr2->name_to, mptr->name_to);
   bzero(mptr2->name_from, sizeof(mptr2->name_from));
   bzero(mptr2->mlen, sizeof(mptr2->mlen));
   sprintf(mptr2->mlen, "%d", 0);
   mptr2->msg[0] = 0x00;

#if FDEBUG
   printf("DEBUG: HNDLADVOK: ABOUT TO SEND %s TO INT %d\n",
          mptr->cmd, lsock);
   printf("DEBUG: HNDLADVOK: MSG SENT = %s %c %s %s %s %s %s %s %s\n",
          mptr2->cmd,
          mptr2->from,
          mptr2->status,
          mptr2->name_to,
          mptr2->name_from,
          mptr2->seqnum,
          mptr2->errno,
          mptr2->mlen,
          mptr2->msg);
#endif

   if (send_msg(lsock, mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR) < 0)
      if (strcmp(mptr2->cmd, "ADVOK") == 1) {
         strcpy(mptr2->cmd, "ADVNO");
         if (namfrmlist(nptr) < 0) {
            printe("HNDLADVOK: BAD RETURN FROM NAMFRMLIST-2\n");
            return (-1);
         }
      }
   ++num;

   if (strcmp(mptr2->cmd, "ADVOK") == 0)
      strcpy(mptr2->cmd, "ADVOKY");
   else {
      strcpy(mptr2->cmd, "ADVOKN");
      close(lsock);
      kill(pid, SIGUSR1);
      FD_CLR(lsock, &ftmtime.readfds_perm);
   }
   strcpy(mptr2->name_from, mptr2->name_to);
   bzero(mptr2->name_to, sizeof(mptr2->name_to));
   for (i = 0; i < ftmsize; ++i)
      if ((ftmptr + i)->sock >= 0)
         if (send_msg((ftmptr + i)->sock, mptr2->cmd,
                      MSGBUF_MIN - MSGBUF_HDR) < 0) {
            if (errno != ENOTCONN) {
               printe("HNDLADVOK: BAD RETURN FROM SEND_MSG-2\n");
               return (-1);
            } else {
               (ftmptr + i)->sock = -3 - (ftmptr + i)->sock;
```

```
            }
         } else
            ++num;
   *delmptr = 'Y';
   return (num + 1);
}
```

```
/*****************************************************************************-
-                                                                            -
-    hndladvoky.c                                                            -
-                                                                            -
-    Purpose - handler routine to implement the second phase of the 2-phase  -
-              commit protocol used to globally advertise names.             -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int              hndladvoky(mptr, mptr2, delmptr)
   struct msgbuf  *mptr,
                  *mptr2;
   char           *delmptr;

/*
 * ------------------------------------------------------------------
 *
 * handle ADVOKY and ADVOKN messages
 *
 * ------------------------------------------------------------------
 */

{
   int             fsock,
                   lsock,
                   pid,
                   msgs,
                   bkupftm;

   struct namtable *nptr;


   if (namlookup(mptr->name_from, &fsock, &lsock, &pid, &bkupftm,
                 &msgs, &nptr) < 0) {
      printe("HNDLADVOKY: BAD RETURN FROM NAMLOOKUP\n");
      return (-1);
   } else if (strcmp(mptr->cmd, "ADVOKY") == 0)
      nptr->status = 'Y';
   else if (namfrmlist(mptr->name_from) < 0) {
      printe("HNDLADVOKY: BAD RETURN FROM NAMFRMLIST\n");
      return (-1);
   }
   *delmptr = 'Y';
```

```
   return (1);
}


/*****************************************************************************
-                                                                           -
-    hndlclose.c                                                            -
-                                                                           -
-    Purpose - handler routine to operate on messages from connected        -
-              applications which attempt to delete an advertised name.     -
-                                                                           -
-****************************************************************************/

#include          "header.h"

int             hndlclose(mptr, mptr2)
   struct msgbuf  *mptr,
                  *mptr2;

/*
 * ------------------------------------------------------------------
 *
 * handle   CLOSE   messages
 *
 * ------------------------------------------------------------------
 */

{
   int             lsock,
                   fsock,
                   pid,
                   msgs,
                   bkupftm;

   struct namtable *nptr;

   extern struct ftm_time ftmtime;


   if (namlookup(mptr->name_from, &lsock, &fsock, &pid, &bkupftm,
                 &msgs, &nptr) < 0)
      return (0);

   if (fsock != -1) {
      printe2("HNDLCLOSE: FSOCK CANNOT BE = %d\n", fsock);
      return (-1);
   }
   bzero(mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR);
   strcpy(mptr2->cmd, "CLOSOK");
   mptr2->from = 'F';
   strcpy(mptr2->status, "-1");
   strcpy(mptr2->name_to, mptr->name_from);
   strcpy(mptr2->seqnum, mptr->seqnum);
```

```
   strcpy(mptr2->errno, "0");
   strcpy(mptr2->mlen, "0");

   if ((send_msg(lsock, mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR)) < 0)
      if (errno != ENOTCONN) {
         perror("HNDLCLOSE: SEND_MSG");
         return (-1);
      }
   close(lsock);
   kill(pid, SIGUSR1);
   FD_CLR(lsock, &ftmtime.readfds_perm);
   nptr->lsock = -2;
   bzero((char *) mptr2, MSGBUF_MIN - MSGBUF_HDR);
   strcpy(mptr2->cmd, "KILLN");
   mptr2->from = 'I';
   strcpy(mptr2->status, "-1");
   strcpy(mptr2->name_from, nptr->name);
   strcpy(mptr2->seqnum, "0");
   strcpy(mptr2->errno, "0");
   sprintf(mptr2->msg, "%X", nptr);
   sprintf(mptr2->mlen, "%d", strlen(mptr2->msg));

   if (msgtolist(mptr2) < 0) {
      printe("HNDLCLOSE: BAD RETURN FROM MSGTOLIST\n");
      errno = EFAULT;
      return (-1);
   }
   return (3);
}
```

```
/*****************************************************************************-
-                                                                            -
-    hndlkilln.c                                                             -
-                                                                            -
-    Purpose - handler routine to operate on messages which indicate that a  -
-              program has died and its advertised name should be deleted.   -
-              The program is then considered for restart/relocation.        -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int             hndlkilln(mptr, mptr2)
   struct msgbuf  *mptr,
                  *mptr2;

/*
 * ------------------------------------------------------------------
 *
 * handle   KILLN   messages
 *
 * ------------------------------------------------------------------
```

```
 */

{
   int             i,
                   err,
                   num,
                   temp1,
                   fsock,
                   temp3,
                   bkupftm = -1,
                   temp5;

   char            name[MAXNAMELEN + 1],
                   bkup[MAXPATHLEN + 1],
                   buf[MAXNAMELEN + MAXHOSTNAMELEN + MAXPATHLEN + 3],
                  *bufpath,
                   from,
                   msgtake;

   FILE           *bkfile;

   struct namtable *nptr;

   struct msgbuf  *mptrtemp;

   extern struct msgbuf *msgfirst;

   extern struct namtable *namfirst;

   extern struct ftmtable *ftmptr;

   extern int      ftmsize,
                   ftmme;


   num = 0;
   sscanf(mptr->errno, "%d", &err);
   strcpy(name, mptr->name_from);
   from = mptr->from;
   nptr = NULL;

   if (name[0] == 0x00) {
      if (mptr->from == 'I')
         sscanf(mptr->msg, "%X", &nptr);
      else {
         printe("HNDLKILLN: MSG FOUND WITH NULL NAME AND FROM != I\n")
            errno = EFAULT;
         return (-1);
      }
   } else
      namlookup(name, &temp1, &fsock, &temp3, &bkupftm, &temp5, &nptr);

   if (nptr != NULL)
      if (namfrmlist(nptr) < 0) {
         printe("HNDLKILLN: BAD RETURN FROM NAMFRMLIST\n");
         return (-1);
      }
   mptr = msgfirst;
   while (mptr != NULL) {
      mptrtemp = mptr->next;
      if (strcmp(mptr->cmd, "ADV") == 0)
         msgtake = 'N';
```

```
        else if (strcmp(mptr->cmd, "TO") == 0)
           if (strcmp(mptr->name_from, name) == 0)
              msgtake = 'Y';
           else if (strcmp(mptr->name_to, name) == 0) {
              sprintf(mptr->status, "%d", -1);
              msgtake = 'N';
           } else
              msgtake = 'N';
        else if (strcmp(mptr->cmd, "TOOK") == 0)
           if (strcmp(mptr->name_to, name) == 0)
              msgtake = 'Y';
           else
              msgtake = 'N';
        else if ((strcmp(mptr->name_to, name) == 0)
                 || (strcmp(mptr->name_from, name) == 0))
           msgtake = 'Y';
        else
           msgtake = 'N';
        if (msgtake == 'Y')
           if (msgfrmlist(mptr) < 0) {
              printe("HNDLKILLN: BAD RETURN FROM MSGFRMLIST\n");
              return (-1);
           } else
              ++num;
        mptr = mptrtemp;
     }

     if ((from == 'I') && ((err == 0) || (err == -1))) {
        bzero(mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR);
        strcpy(mptr2->cmd, "KILLN");
        mptr2->from = 'F';
        strcpy(mptr2->status, "-1");
        strcpy(mptr2->name_from, name);
        strcpy(mptr2->seqnum, "0");
        sprintf(mptr2->errno, "%d", err);
        strcpy(mptr2->mlen, "0");
        for (i = 0; i < ftmsize; ++i)
           if (((ftmptr + i)->sock >= 0) && ((ftmptr + i)->sock != fsock))
              if (send_msg
                ((ftmptr + i)->sock, mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR) < 0) {
                 if (errno != ENOTCONN) {
                    perror("HNDLKILLN: SEND_MSG-1");
                    return (-1);
                 } else {
                    (ftmptr + i)->sock = -3 - (ftmptr + i)->sock;
                 }
              } else
                 ++num;
     }
     if (err == 0)
        return (num);

     if (bkupftm != ftmme)
        return (num);

  #if FDEBUG
     printf("DEBUG: HNDLKILLN: ATTEMPTING TO RESTART NAME = %s\n", name);
  #endif

     bzero(bkup, sizeof(bkup));
     strcpy(bkup, FTMPATH);
     strcat(bkup, BKUPFILE);
```

```
   if ((bkfile = fopen(bkup, "r")) == NULL) {
      printe("FTM: HNDLKILLN: NO BACKUP FILE FOUND\n");
      return (0);
   }
   bzero(buf, sizeof(buf));
   while (fgets(buf, sizeof(buf), bkfile) != NULL) {
      buf[sizeof(buf) - 1] = 0x00;
      nline(buf, sizeof(buf));
      for (bufpath = buf; bufpath[0] != 0x00 && bufpath[0] != TAB; ++bufpath);
      if (bufpath[0] == 0x00) { /* NO TAB ==> COMMENT CARD */
         bzero(buf, sizeof(buf));
         continue;
      }
      bufpath[0] = 0x00;
      setcap(buf, strlen(buf));

#if FDEBUG
      printf("DEBUG: HNDLKILLN: ADV NAME = %s   FILE LINE NAME = %s\n", name, bu
f);
#endif

      if (strcmp(buf, name) != 0) {     /* DIFFERENT ADVERTISED NAME */
         bzero(buf, sizeof(buf));
         continue;
      }
      ++bufpath;
      while (bufpath[0] == TAB)
         ++bufpath;

#if FDEBUG
      printf("DEBUG: HNDLKILLN: RESTART NAME MATCH FOUND,/n   PATH = %s\n",
            bufpath);
#endif

      bzero(mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR);
      strcpy(mptr2->cmd, "KILLN");
      mptr2->from = 'F';
      strcpy(mptr2->status, "-1");
      strcpy(mptr2->name_from, name);
      strcpy(mptr2->seqnum, "0");
      sprintf(mptr2->errno, "%d", err);
      strcpy(mptr2->mlen, "0");
      for (i = 0; i < ftmsize; ++i)
         if (((ftmptr + i)->sock >= 0) && ((ftmptr + i)->sock != fsock))
            if (send_msg
              ((ftmptr + i)->sock, mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR) < 0) {
               if (errno != ENOTCONN) {
                  perror("HNDLKILLN: SEND_MSG-2");
                  return (-1);
               } else {
                  (ftmptr + i)->sock = -3 - (ftmptr + i)->sock;
               }
            } else
               ++num;

      if ((i = fork()) < 0) {
         perror("FTM: HNDLKILLN: FORK");
         printe("FTM: HNDLKILLN: COULD NOT FORK A PROCESS RESTART\n");
         break;
      } else if (i > 0)
         break;
```

```
        else {
            ++num;
            execve(bufpath, 0, 0);
            perror("FTM CHILD: HNDLKILLN: EXECVE");
            printe2("FTM CHILD: HNDLKILLN: COULD NOT RESTART PROCESS = %s\n",
                    bufpath);
            exit(-1);
        }
    }

    fclose(bkfile);
    return (num);
}


/*****************************************************************************
-                                                                           -
-    hndlstatus.c                                                           -
-                                                                           -
-    Purpose - handler routine to operate on messages from applications which -
-              request global status.                                       -
-                                                                           -
-****************************************************************************/

#include          "header.h"

int             hndlstatus(mptr, mptr2)
    struct msgbuf  *mptr,
                   *mptr2;

/*
 * ------------------------------------------------------------------
 *
 * handle   STATUS   messages
 *
 * ------------------------------------------------------------------
 */

{
    int             i,
                    sunstat,
                    mirror,
                    lsock,
                    fsock,
                    pid,
                    msgs,
                    bkupftm;

    char            mach[MAXHOSTNAMELEN + 1];

    struct namtable *nptr;

    extern struct ftmtable *ftmptr;
```

```
   extern int      ftmsize;

/*
sunstat: BIT 0  => auvsun0
         BIT 1  => auvc1
         BIT 2  => auvc2
         BIT 3  => auvc3
         BIT 4  => auvc4
         BIT 5  => auvc5
         BIT 6  => auvc6
         BIT 7  => auvc7
         BIT 8  => auvc8
         BIT 9  => auvc9
         BIT 10 => auvc10
         BIT 11 => auvc11
         BIT 12 -> auvc12
         BIT 13 => auvc13
         BIT 14 => auvc14
         BIT 15 => auvc15
         BIT 16 => auvc16
         BIT 17 => auvsun1

mirror == 0 => DRIVE 0 DOWN   *** MEANINGLESS UNTIL DISK MIRRORING INSTALLED
       == 1 => DRIVE 1 DOWN
       == 2 => BOTH DRIVES UP
*/


   mirror = 2;

   sunstat = 0;
   for (i = 0; i < ftmsize; ++i) {
      if ((ftmptr + i)->sock < -1)
         continue;
      strcpy(mach, (ftmptr + i)->mach);
      if (strcmp(mach, "auvsun0") == 0)
         sunstat |= (1 << 0);
      else if (strcmp(mach, "auvc1") == 0)
         sunstat |= (1 << 1);
      else if (strcmp(mach, "auvc2") == 0)
         sunstat |= (1 << 2);
      else if (strcmp(mach, "auvc3") == 0)
         sunstat |= (1 << 3);
      else if (strcmp(mach, "auvc4") == 0)
         sunstat |= (1 << 4);
      else if (strcmp(mach, "auvc5") == 0)
         sunstat |= (1 << 5);
      else if (strcmp(mach, "auvc6") == 0)
         sunstat |= (1 << 6);
      else if (strcmp(mach, "auvc7") == 0)
         sunstat |= (1 << 7);
      else if (strcmp(mach, "auvc8") == 0)
         sunstat |= (1 << 8);
      else if (strcmp(mach, "auvc9") == 0)
         sunstat |= (1 << 9);
      else if (strcmp(mach, "auvc10") == 0)
         sunstat |= (1 << 10);
      else if (strcmp(mach, "auvc11") == 0)
         sunstat |= (1 << 11);
      else if (strcmp(mach, "auvc12") == 0)
         sunstat |= (1 << 12);
```

```
      else if (strcmp(mach, "auvc13") == 0)
         sunstat |= (1 << 13);
      else if (strcmp(mach, "auvc14") == 0)
         sunstat |= (1 << 14);
      else if (strcmp(mach, "auvc15") == 0)
         sunstat |= (1 << 15);
      else if (strcmp(mach, "auvc16") == 0)
         sunstat |= (1 << 16);
      else if (strcmp(mach, "auvsun2") == 0)
         sunstat |= (1 << 17);
   }

   bzero(mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR);
   strcpy(mptr2->cmd, "STATOK");
   mptr2->from = 'F';
   strcpy(mptr2->status, "-1");
   strcpy(mptr2->name_to, mptr->name_from);
   strcpy(mptr2->seqnum, mptr->seqnum);
   strcpy(mptr2->errno, "0");
   sprintf(mptr2->msg, "%X %X", sunstat, mirror);
   sprintf(mptr2->mlen, "%d", strlen(mptr2->msg));

   if ((namlookup(mptr2->name_to, &lsock, &fsock, &pid, &bkupftm, &msgs,
                  &nptr) < 0) || (fsock != -1) || (lsock < 0))
      return (0);              /* THE APPLICATION HAS GONE AWAY */
   else if ((send_msg(lsock,
               mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR + strlen(mptr2->msg))) < 0)
      if (errno != ENOTCONN) {
         perror("HNDLSTATUS: SEND_MSG");
         return (-1);
      }
   return (2);
}
```

```
/*****************************************************************************
-                                                                           -
-    hndlto.c                                                               -
-                                                                           -
-    Purpose - handler routine to operate on messages which are to be sent  -
-              to another FTM or directly to an application.                -
-                                                                           -
-****************************************************************************/

#include          "header.h"

int             hndlto(mptr, mptr2, delmptr)
   struct msgbuf  *mptr,
                  *mptr2;
   char           *delmptr;

/*
 * ---------------------------------------------------------------------
```

```
 *
 * handle   TO   messages
 *
 * ---------------------------------------------------------------------
 */

{
   int              s,
                    i,
                    mptrlen,
                    lsock,
                    fsock,
                    pid,
                    msgs,
                    num,
                    bkupftm;

   char             ftmloc;

   struct namtable *nptr;

   extern struct ftmtable *ftmptr;

   extern int       ftmsize;


   *delmptr = 'N';
   sscanf(mptr->mlen, "%d", &mptrlen);
   mptrlen += MSGBUF_MIN;
   num = 0;

   if ((namlookup(mptr->name_to, &lsock, &fsock, &pid, &bkupftm, &msgs,
                  &nptr) < 0)
       || (nptr->status == 'N') || ((fsock == -1) && (lsock < 0))) {
      bzero(mptr2->cmd, mptrlen - MSGBUF_MIN);
      strcpy(mptr2->cmd, "TONO");
      mptr2->from = 'F';
      strcpy(mptr2->status, mptr->status);
      strcpy(mptr2->name_to, mptr->name_from);
      strcpy(mptr2->name_from, mptr->name_to);
      strcpy(mptr2->seqnum, mptr->seqnum);
      sprintf(mptr2->errno, "%d", EWOULDBLOCK);
      strcpy(mptr2->mlen, mptr->mlen);
      bcopy(mptr->msg, mptr2->msg, mptrlen - MSGBUF_MIN + 1);
      if (namlookup(mptr->name_from, &lsock, &fsock, &pid, &bkupftm,
                    &msgs, &nptr) < 0) {
         *delmptr = 'Y';
         return (0);
      }
      if (fsock == -1) {
         s = lsock;
         ftmloc = 'L';
      } else if (fsock >= 0) {
         s = fsock;
         ftmloc = 'F';
      } else {
         printe3("HNDLTO: LOGIC ERROR TO(1), FSOCK = %d FOR NAME %s\n",
                 fsock, mptr->name_from);
         return (-1);
      }
      if (s >= 0) {
         if (send_msg(s, mptr2->cmd, mptrlen - MSGBUF_HDR) < 0) {
```

```
            if (errno != ENOTCONN) {
               perror("HNDLTO: SEND_MSG-1");
               printe2("HNDLTO: ATTEMPT TO SEND TO SOCKET = %d\n", s);
               errno = EFAULT;
               return (-1);
            } else if (ftmloc == 'F')
               for (i = 0; i < ftmsize; ++i)
                  if ((ftmptr + i)->sock == s) {
                     (ftmptr + i)->sock = -3 - s;
                     break;
                  }
            *delmptr = 'Y';
            return (1);
         } else {

#if FDEBUG
            printf("DEBUG: HNDLTO: TONO MESSAGE SENT TO SOCK = %d\n", s);
            printf("DEBUG: HNDLTO: MSG SENT = %s %c %s %s %s %s %s %s %s\n",
                   mptr2->cmd,
                   mptr2->from,
                   mptr2->status,
                   mptr2->name_to,
                   mptr2->name_from,
                   mptr2->seqnum,
                   mptr2->errno,
                   mptr2->mlen,
                   mptr2->msg);
#endif

         }
         *delmptr = 'Y';
      }
   } else if ((fsock == -1) || (fsock >= 0)) {
      if (fsock == -1) {
         s = lsock;
         ftmloc = 'L';
      } else {
         s = fsock;
         ftmloc = 'F';
      }
      bzero(mptr2->cmd, mptrlen - MSGBUF_HDR);
      strcpy(mptr2->cmd, "TO");
      mptr2->from = 'F';
      sprintf(mptr2->status, "%d", -1);
      strcpy(mptr2->name_to, mptr->name_to);
      strcpy(mptr2->name_from, mptr->name_from);
      strcpy(mptr2->seqnum, mptr->seqnum);
      sprintf(mptr2->errno, "%d", 0);
      strcpy(mptr2->mlen, mptr->mlen);
      bcopy(mptr->msg, mptr2->msg, mptrlen - MSGBUF_MIN + 1);

      if (send_msg(s, mptr2->cmd, mptrlen - MSGBUF_HDR) < 0) {
         if (errno != ENOTCONN) {
            perror("HNDLTO: SEND_MSG-2");
            printe2("HNDLTO: ATTEMPT TO SEND TO SOCKET = %d\n", s);
            errno = EFAULT;
            return (-1);
         } else if (ftmloc == 'F')
            for (i = 0; i < ftmsize; ++i)
               if ((ftmptr + i)->sock == s) {
                  (ftmptr + i)->sock = -3 - s;
                  break;
```

```
            }
        *delmptr = 'Y';
        return (1);
     } else {

#if FDEBUG
        printf("DEBUG: HNDLTO: TO MESSAGE SENT TO SOCK = %d\n", s);
        printf("DEBUG: HNDLTO: MSG SENT = %s %c %s %s %s %s %s %s %s\n",
               mptr2->cmd,
               mptr2->from,
               mptr2->status,
               mptr2->name_to,
               mptr2->name_from,
               mptr2->seqnum,
               mptr2->errno,
               mptr2->mlen,
               mptr2->msg);
#endif

        bzero(mptr->status, sizeof(mptr->status));
        sprintf(mptr->status, "%d", 1);
        ++num;
     }
   } else {
     printe3("HNDLTO: LOGIC ERROR TO(2), FSOCK = %d FOR NAME %s\n",
             fsock, mptr->name_to);
     return (-1);
   }

   return (num + 1);
}
```

```
/*****************************************************************************
-                                                                           -
-    hndltook.c                                                             -
-                                                                           -
-    Purpose - handler routine to operate on messages from the receiver of a -
-              previous message, indicating (un)successful receipt.         -
-                                                                           -
-****************************************************************************/

#include          "header.h"

int             hndltook(mptr, mptr2, delmptr)
   struct msgbuf  *mptr,
                  *mptr2;
   char           *delmptr;

/*
 * ---------------------------------------------------------------------
 *
 * handle   TOOK and TONO   messages
```

```
 *
 * ----------------------------------------------------------------------
 */

{
   int              s,
                    i,
                    mptrlen,
                    lsock,
                    fsock,
                    pid,
                    msgs,
                    num,
                    status,
                    bkupftm;

   char             ftmloc;

   struct namtable *nptr;

   extern struct ftmtable *ftmptr;

   extern int       ftmsize;


   *delmptr = 'N';
   sscanf(mptr->mlen, "%d", &mptrlen);
   mptrlen += MSGBUF_MIN;
   num = 0;
   if ((namlookup(mptr->name_to, &lsock, &fsock, &pid, &bkupftm,
                  &msgs, &nptr) < 0) || ((fsock == -1) && (lsock < 0))) {
      *delmptr = 'Y';
      return (0);
   } else if (findbuf("TO", mptr->name_to, mptr2) < 0) {
      printe("HNDLTOOK: CANNOT FIND A TO MESSAGE\n");
      return (-1);
   } else if (strcmp(mptr->seqnum, mptr2->seqnum) != 0) {
      printe3("HNDLTOOK: SEQNUM MISMATCH %s %s\n", mptr2->seqnum,
              mptr->seqnum);
      return (-1);
   } else if (strcmp(mptr->mlen, mptr2->mlen) != 0) {
      printe3("HNDLTOOK: MESSAGE LENGTH MISMATCH %s %s\n", mptr2->mlen,
              mptr->mlen);
      return (-1);
   } else if (bcmp(mptr->msg, mptr2->msg, mptrlen - MSGBUF_MIN) != 0) {
      printe("HNDLTOOK: MESSAGE GARBLED DURING TRANSMISSION:\n");
      msgdump("MPTR", mptr);
      msgdump("MPTR2", mptr2);
      return (-1);
   } else {
      sscanf(mptr2->status, "%d", &status);
      --status;
      if (status > 0) {
         bzero(mptr2->status, sizeof(mptr2->status));
         sprintf(mptr2->status, "%d", status);
         if (msgtolist(mptr2) < 0) {
            printe("HNDLTOOK: BAD RETURN FROM MSGTOLIST\n");
            return (-1);
         } else {
            *delmptr = 'Y';
            return (num + 1);
         }
```

```
        } else {
            ++num;
        }
    }

    if (fsock == -1) {
        s = lsock;
        ftmloc = 'L';
    } else {
        s = fsock;
        ftmloc = 'F';
    }
    bzero(mptr2->cmd, mptrlen - MSGBUF_HDR);
    strcpy(mptr2->cmd, "TOOK");
    mptr2->from = 'F';
    sprintf(mptr2->status, "%d", -1);
    strcpy(mptr2->name_to, mptr->name_to);
    strcpy(mptr2->name_from, mptr->name_from);
    strcpy(mptr2->seqnum, mptr->seqnum);
    sprintf(mptr2->errno, "%d", 0);
    sprintf(mptr2->mlen, "%d", mptrlen - MSGBUF_MIN);
    bcopy(mptr->msg, mptr2->msg, mptrlen - MSGBUF_HDR);
    if (send_msg(s, mptr2->cmd, mptrlen - MSGBUF_HDR) < 0) {
        if (errno != ENOTCONN) {
            perror("HNDLTOOK: SEND_MSG-1");
            printe2("HNDLTOOK: ATTEMPT TO SEND TO SOCKET = %d\n", s);
            errno = EFAULT;
            return (-1);
        } else if (ftmloc == 'F')
            for (i = 0; i < ftmsize; ++i)
                if ((ftmptr + i)->sock == s) {
                    (ftmptr + i)->sock = -3 - s;
                    break;
                }
        *delmptr = 'Y';
        return (1);
    } else {

#if FDEBUG
        printf("DEBUG: HNDLTOOK: MESSAGE SENT TO SOCK = %d\n", s);
        printf("DEBUG: HNDLTOOK: MSG SENT = %s %c %s %s %s %s %s %s %s\n",
                mptr2->cmd,
                mptr2->from,
                mptr2->status,
                mptr2->name_to,
                mptr2->name_from,
                mptr2->seqnum,
                mptr2->errno,
                mptr2->mlen,
                mptr2->msg);
#endif

        ;
    }

    *delmptr = 'Y';
    return (num + 1);
}
```

```
/*****************************************************************************
-                                                                            -
-    hndlwhere.c                                                             -
-                                                                            -
-    Purpose - handler routine to operate on messages which inquire about the -
-              current and backup systems of any currently executing         -
-              application.                                                  -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int             hndlwhere(mptr, mptr2)
   struct msgbuf  *mptr,
                  *mptr2;

/*
 * ---------------------------------------------------------------------
 *
 * handle   WHERE   messages
 *
 * ---------------------------------------------------------------------
 */

{
   int             i,
                   lsock,
                   fsock,
                   pid,
                   msgs,
                   bkupftm;

   char            mach[MAXHOSTNAMELEN + 1],
                   bkup[MAXHOSTNAMELEN + 1];

   struct namtable *nptr;

   extern struct ftmtable *ftmptr;

   extern int      ftmsize,
                   ftmme;


   mach[0] = 0x00;
   bkup[0] = 0x00;
   if ((namlookup(mptr->msg, &lsock, &fsock, &pid, &bkupftm,
                  &msgs, &nptr) >= 0) && (nptr->status == 'Y')) {
      if (fsock == -1)
         if (lsock >= 0)
            strcpy(mach, (ftmptr + ftmme)->mach);
         else;
      else
         for (i = 0; i < ftmsize; ++i)
            if ((ftmptr + i)->sock == fsock) {
               strcpy(mach, (ftmptr + i)->mach);
               break;
            }
      strcpy(bkup, (ftmptr + bkupftm)->mach);
   }
   bzero(mptr2->cmd, MSGBUF_MIN - MSGBUF_HDR);
   strcpy(mptr2->cmd, "WHEROK");
   mptr2->from = 'F';
```

```
   strcpy(mptr2->status, "-1");
   strcpy(mptr2->name_to, mptr->name_from);
   strcpy(mptr2->seqnum, mptr->seqnum);
   strcpy(mptr2->errno, "0");
   bcopy(mach, mptr2->msg, MAXHOSTNAMELEN + 1);
   bcopy(bkup, (mptr2->msg) + MAXHOSTNAMELEN + 1, MAXHOSTNAMELEN + 1);
   sprintf(mptr2->mlen, "%d", 2 * (MAXHOSTNAMELEN + 1));

   if ((namlookup(mptr2->name_to, &lsock, &fsock, &pid, &bkupftm,
                  &msgs, &nptr) < 0) || (fsock != -1) || (lsock < 0))
      return (1);               /* THE APPLICATION WENT AWAY */
   else if ((send_msg(lsock,
                      mptr2->cmd,
                      MSGBUF_MIN - MSGBUF_HDR + 2 * (MAXHOSTNAMELEN + 1)))
            < 0)
      if (errno != ENOTCONN) {
         perror("HNDLWHERE: SEND_MSG");
         return (-1);
      }
   return (2);
}
```

```
/*****************************************************************************
-                                                                           -
-    ifget.c                                                                -
-                                                                           -
-    Purpose - called by the FTM to gather pending messages from locally -
-              connected applications.                                      -
-                                                                           -
-****************************************************************************/

#include            "header.h"

int             ifget()
{
   int             s,
                   pid,
                   thispass,
                   allpasses;

   struct msgbuf  *mptr2;

   struct namtable *nptr;

   extern struct namtable *namfirst;

   extern struct ftm_time ftmtime.


   if ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL) {
      perror("IFGET: MALLOC");
      errno = EFAULT;
```

```
        return (-1);
    }
    allpasses = 0;
    LOOP {
        thispass = 0;
        nptr = namfirst;
        for (nptr = namfirst; nptr != NULL; nptr = nptr->next) {
            if (nptr->fsock != -1)
                continue;
            if (nptr->lsock < 0)
                continue;
            s = nptr->lsock;
            if (recv_msg(s, mptr2->cmd, MSGBUF_MAX - MSGBUF_HDR) < 0) {
                if (errno == EWOULDBLOCK)
                    continue;
                else if (errno == ENOTCONN) {

#if FDEBUG
                    printf("DEBUG: IFGET: SOCKET CLOSED = %d   NAME = %s\n",
                           s, nptr->name);
#endif

                    close(s);
                    FD_CLR(s, &ftmtime.readfds_perm);
                    nptr->lsock = -2;
                    nptr->status = 'R';
                    bzero((char *) mptr2, MSGBUF_MIN - MSGBUF_HDR);
                    strcpy(mptr2->cmd, "KILLN");
                    mptr2->from = 'I';
                    strcpy(mptr2->status, "-1");
                    strcpy(mptr2->name_from, nptr->name);
                    strcpy(mptr2->seqnum, "0");
                    sprintf(mptr2->errno, "%d", -1);
                    sprintf(mptr2->msg, "%X", nptr);
                    sprintf(mptr2->mlen, "%d", strlen(mptr2->msg));
                } else {
                    perror("IFGET: RECV_MSG");
                    free((char *) mptr2);
                    errno = EFAULT;
                    return (-1);
                }
            }

#if FDEBUG
            printf("DEBUG: IFGET: MSG FOUND = %s %c %s %s %s %s %s %s %s\n",
                   mptr2->cmd,
                   mptr2->from,
                   mptr2->status,
                   mptr2->name_to,
                   mptr2->name_from,
                   mptr2->seqnum,
                   mptr2->errno,
                   mptr2->mlen,
                   mptr2->msg);
#endif

            if (strcmp(mptr2->cmd, "ADV") == 0) {
                sscanf(mptr2->msg, "%d", &pid);
                sprintf(mptr2->msg, "%d %d %d", pid, s, pick_bkup());
                sprintf(mptr2->mlen, "%d", strlen(mptr2->msg));
            }
            if (msgtolist(mptr2) < 0) {
```

```
            printe("IFGET: BAD RETURN FROM MSGTOLIST\n");
            free((char *) mptr2);
            errno = EFAULT;
            return (-1);
         }
         ++thispass;
      }
      allpasses += thispass;
/* ----- ONLY ALLOW ONE MESSAGE PER SOCKET PER FTM MAIN LOOP PASS ----- **
      if (thispass == 0)
*/
      break;
   }

   free((char *) mptr2);
   return (allpasses);
}
```

```
/*************************************************************************-
-                                                                        -
-    msgdump.c                                                           -
-                                                                        -
-    Purpose - formats a dump of a message which includes a HEX dump of the -
-              MSG part to compare for garbled transmissions.            -
-                                                                        -
-*************************************************************************/

#include          "header.h"

void            msgdump(string, mptr)
   char           *string;
   struct msgbuf  *mptr;

{
   int             i,
                   j,
                   k,
                   numbytes;

   char            line[81];


   printe2("MSGDUMP: %s\n", string);
   printe9("MSGDUMP: HEADER = %s %c %s %s %s %s %s %s\n",
           mptr->cmd, mptr->from, mptr->status, mptr->name_to,
           mptr->name_from, mptr->seqnum, mptr->errno, mptr->mlen);
   printe2("MSGDUMP: MSG = %s\n", mptr->msg);
   printe("MSGDUMP: MESSAGE IN HEX FOLLOWS:\n");

   sscanf(mptr->mlen, "%d", &numbytes);
   i = 0;
```

```
   k = 0;
   bzero(line, 81);

   while (i < numbytes) {
      j = (int) mptr->msg[i];
      sprintf(line + k, "%02X", j);
      ++i;
      k += 2;
      if (k == 78) {
         printc2("%s\n", line);
         bzero(line, 80);
         k = 0;
      }
   }

   if (k != 0)
      printe2("%s\n", line);
   printe("\n");

   return;
}
```

```
/*****************************************************************************
-                                                                            -
-    msgfrmlist.c                                                            -
-                                                                            -
-    Purpose - deletes a given message buffer from the current message list. -
-                                                                            -
-****************************************************************************/

#include            "header.h"

int             msgfrmlist(mptr)
   struct msgbuf  *mptr;

{
   struct msgbuf  *mptr2;

   extern struct msgbuf *msgfirst,
                  *msglast;


   for (mptr2 = msgfirst; mptr2 != NULL; mptr2 = mptr2->next)
      if (mptr2 == mptr)
         break;

   if (mptr2 == NULL) {
      printe("MSGFRMLIST: BUFFER TO DELETE NOT ON LIST\n");
      return (-1);
   }
   if (msgfirst == msglast) {   /* ONLY MSG ON QUEUE     */
      msgfirst = NULL;
```

```
      msglast = NULL;
   } else if (mptr == msgfirst) {        /* HEAD OF QUEUE          */
      msgfirst = msgfirst->next;
      msgfirst->prev = NULL;
   } else if (mptr == msglast) {/* TAIL OF QUEUE          */
      msglast = mptr->prev;
      msglast->next = NULL;
   } else {                      /* MIDDLE OF QUEUE        */
      mptr->prev->next = mptr->next;
      mptr->next->prev = mptr->prev;
   }

   free((char *) mptr);

   return (0);
}
```

```
/*************************************************************************
-                                                                        -
-    msghndl.c                                                           -
-                                                                        -
-    Purpose - steps through the entire linked list of pending FTM messages.  -
-              Ignores or modifies messages, or calls hndlXXXXX () routines   -
-              as appropriate.                                             -
-                                                                        -
-*************************************************************************/

#include          "header.h"

int             msghndl()
{
   int             num,
                   totnum,
                   status;

   char            delmptr;

   struct msgbuf  *mptr,
                  *mptr2,
                  *mptrtemp;

   extern struct msgbuf *msgfirst;


   if ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL) {
      perror("MSGHNDL: MALLOC");
      return (-1);
   }
   totnum = 0;
   mptr = msgfirst;
   while (mptr != NULL) {
      sscanf(mptr->status, "%d", &status);
```

```
if (status == 0) {
   printe("MSGHNDL: MSG FOUND WITH STATUS = 0\n");
   printe9
      ("MSGHNDL: FULL MESSAGE IN ERROR  = %s %c %s %s %s %s %s %s %s\n",
       mptr->cmd, mptr->from, mptr->status, mptr->name_to,
       mptr->name_from, mptr->seqnum, mptr->errno, mptr->mlen);
   return (-1);
}
if (status > 0) {
   mptr = mptr->next;
   continue;
}
if ((strcmp(mptr->cmd, "ADV") == 0) && (mptr->from == 'F')) {
   if ((num = hndladvf(mptr, mptr2, &delmptr)) < 0) {
      printe("MSGHNDL: BAD RETURN FROM HNDLADVF\n");
      free((char *) mptr2);
      return (-1);
   } else
      totnum += num;
} else if ((strcmp(mptr->cmd, "ADV") == 0) && (mptr->from == 'I')) {
   if ((num = hndladvi(mptr, mptr2, &delmptr)) < 0) {
      printe("MSGHNDL: BAD RETURN FROM HNDLADVI\n");
      free((char *) mptr2);
      return (-1);
   } else
      totnum += num;
} else if ((strcmp(mptr->cmd, "ADVOK") == 0) ||
           (strcmp(mptr->cmd, "ADVNO") == 0)) {
   if ((num = hndladvok(mptr, mptr2, &delmptr)) < 0) {
      printe("MSGHNDL: BAD RETURN FROM HNDLADVOK\n");
      free((char *) mptr2);
      return (-1);
   } else
      totnum += num;
} else if ((strcmp(mptr->cmd, "ADVOKY") == 0) ||
           (strcmp(mptr->cmd, "ADVOKN") == 0)) {
   if ((num = hndladvoky(mptr, mptr2, &delmptr)) < 0) {
      printe("MSGHNDL: BAD RETURN FROM HNDLADVOKY\n");
      free((char *) mptr2);
      return (-1);
   } else
      totnum += num;
} else if (strcmp(mptr->cmd, "CLOSE") == 0) {
   if ((num = hndlclose(mptr, mptr2)) < 0) {
      printe("MSGHNDL: BAD RETURN FROM HNDLCLOSE\n");
      free((char *) mptr2);
      return (-1);
   } else {
      totnum += num;
      delmptr = 'Y';
   }
} else if (strcmp(mptr->cmd, "KILLN") == 0) {
   if ((num = hndlkilln(mptr, mptr2)) < 0) {
      printe("MSGHNDL: BAD RETURN FROM HNDLKILLN\n");
      free((char *) mptr2);
      return (-1);
   } else {
      totnum += num;
      mptr = msgfirst;
      delmptr = 'N';
   }
} else if (strcmp(mptr->cmd, "STATUS") == 0) {
```

```
            if ((num = hndlstatus(mptr, mptr2)) < 0) {
               printe("MSGHNDL: BAD RETURN FROM HNDLSTATUS\n");
               free((char *) mptr2);
               return (-1);
            } else {
               totnum += num;
               delmptr = 'Y';
            }
         } else if (strcmp(mptr->cmd, "TO") == 0) {
            if ((num = hndlto(mptr, mptr2, &delmptr)) < 0) {
               printe("MSGHNDL: BAD RETURN FROM HNDLTO\n");
               free((char *) mptr2);
               return (-1);
            } else
               totnum += num;
         } else if ((strcmp(mptr->cmd, "TOOK") == 0) ||
                    (strcmp(mptr->cmd, "TONO") == 0)) {
            if ((num = hndltook(mptr, mptr2, &delmptr)) < 0) {
               printe("MSGHNDL: BAD RETURN FROM HNDLTOOK\n");
               free((char *) mptr2);
               return (-1);
            } else
               totnum += num;
         } else if (strcmp(mptr->cmd, "WHERE") == 0) {
            if ((num = hndlwhere(mptr, mptr2)) < 0) {
               printe("MSGHNDL: BAD RETURN FROM HNDLWHERE\n");
               free((char *) mptr2);
               return (-1);
            } else {
               totnum += num;
               delmptr = 'Y';
            }
         } else {
            printe9
               ("MSGHNDL: UNKNOWN ACTION MESSAGE = %s %c %s %s %s %s %s %s %s\n",
                mptr->cmd, mptr->from, mptr->status, mptr->name_to,
                mptr->name_from, mptr->seqnum, mptr->errno, mptr->mlen);
            printe("MSGHNDL: ABOVE MESSAGE WILL BE DELETED\n");
            delmptr = 'Y';
            ++totnum;
         }

         if (mptr != NULL) {
            mptrtemp = mptr;
            mptr = mptr->next;
            if (delmptr == 'Y')
               if (msgfrmlist(mptrtemp) < 0) {
                  printe("MSGHNDL: BAD RETURN FROM MSGFRMLIST\n");
                  return (-1);
               }
         }
      }

      free((char *) mptr2);
      return (totnum);
   }
```

```
/*****************************************************************************-
-                                                                            -
-    msgtolist.c                                                             -
-                                                                            -
-    Purpose - inserts a message into the linked message list.               -
-                                                                            -
-*****************************************************************************/

#include            "header.h"

int             msgtolist(mybuf)
   struct msgbuf  *mybuf;

{
   int             msglen;

   struct msgbuf  *mptr1;

   extern struct msgbuf *msgfirst,
                  *msglast;


   sscanf(mybuf->mlen, "%d", &msglen);
   msglen += MSGBUF_MIN;

   if ((mptr1 = (struct msgbuf *) malloc(msglen)) == NULL) {
      perror("MSGTOLIST: MALLOC");
      printe2("MSGTOLIST: MSGLEN = %d\n", msglen);
      return (-1);
   }
   bcopy((char *) mybuf, (char *) mptr1, msglen);
   mptr1->next = NULL;
   mptr1->prev = NULL;

   if (msglast == NULL) {
      msgfirst = mptr1;
      msglast = mptr1;
   } else {
      msglast->next = mptr1;
      mptr1->prev = msglast;
      msglast = mptr1;
   }

   return (0);
}
```

```
/*****************************************************************************
-                                                                            -
-    namfrmlist.c                                                            -
-                                                                            -
-    Purpose - removes an advertised name from the local list of global      -
-              advertised names.                                             -
-                                                                            -
-****************************************************************************/

#include            "header.h"

int             namfrmlist(nptr)
   struct namtable *nptr;

{
   struct namtable *nptr2;

   extern struct namtable *namfirst,
                  *namlast;


   for (nptr2 = namfirst; nptr2 != NULL; nptr2 = nptr2->next)
      if (nptr2 == nptr)
         break;

   if (nptr2 == NULL) {
      printe("NAMFRMLIST: BUFFER TO DELETE NOT ON LIST\n");
      return (-1);
   }
   if (namfirst == namlast) {   /* ONLY NAM ON QUEUE     */
      namfirst = NULL;
      namlast = NULL;
   } else if (nptr == namfirst) {       /* HEAD OF QUEUE         */
      namfirst = namfirst->next;
      namfirst->prev = NULL;
   } else if (nptr == namlast) {/* TAIL OF QUEUE         */
      namlast = nptr->prev;
      namlast->next = NULL;
   } else {                     /* MIDDLE OF QUEUE       */
      nptr->prev->next = nptr->next;
      nptr->next->prev = nptr->prev;
   }

   free((char *) nptr);

   return (0);
}
```

```
/*****************************************************************************-
-                                                                            -
-    namlookup.c                                                             -
-                                                                            -
-    Purpose - gets information about a globally advertised name from the    -
-              local table of global names.                                  -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int               namlookup(name, lsock, fsock, pid, bkupftm, msgs, nptrparm)
   char            *name;
   int             *lsock,
                   *fsock,
                   *pid,
                   *msgs,
                   *bkupftm;
   struct namtable **nptrparm;


{
   struct namtable *nptr;

   extern struct namtable *namfirst;


   for (nptr = namfirst; nptr != NULL; nptr = nptr->next)
      if (name[0] != 0x00)
         if (strcmp(nptr->name, name) == 0)
            break;
         else
            continue;
      else if ((nptr->fsock == -1) && (nptr->lsock == *lsock))
         break;

   if (nptr == NULL)
      return (-1);
   else {
      *lsock = nptr->lsock;
      *fsock = nptr->fsock;
      *pid = nptr->pid;
      *bkupftm = nptr->bkupftm;
      *msgs = nptr->msgs;
      *nptrparm = nptr;
      return (0);
   }
}
```

```
/*****************************************************************************-
-                                                                            -
-    namtolist.c                                                             -
-                                                                            -
-    Purpose - adds an entry into the table (actually it is a linked list) of -
-              globally advertised names.                                    -
-                                                                            -
-****************************************************************************/

#include          "header.h"

int             namtolist(name, lsock, fsock, pid, bkupftm)
   char           *name;
   int             lsock,
                   fsock,
                   pid,
                   bkupftm;


{
   struct namtable *nptr;

   extern struct namtable *namfirst,
                  *namlast;


   errno = EFAULT;

   if (lsock < 0) {
      printe2("NAMTOLIST: ILLEGAL LSOCK = %d\n", lsock);
      return (-1);
   }
   if ((fsock < 0) && (fsock != -1)) {
      printe2("NAMTOLIST: ILLEGAL FSOCK = %d\n", fsock);
      return (-1);
   }
   if ((fsock >= 0) && (name[0] == 0x00)) {
      printe2("NAMTOLIST: NAME CANNOT BE NULL FOR FSOCK = %d\n", fsock);
      return (-1);
   }
 /*
  * -   END OF ERRORS
  */

   if (name[0] != 0x00)
      for (nptr = namfirst; nptr != NULL; nptr = nptr->next)
         if (strcmp(nptr->name, name) == 0) {
            errno = EWOULDBLOCK;
            return (-1);
         }
   if ((fsock == -1) && (name[0] != 0x00)) {
      for (nptr = namfirst; nptr != NULL; nptr = nptr->next)
         if ((nptr->lsock == lsock) && (nptr->fsock == fsock)) {
            strcpy(nptr->name, name);
            nptr->pid = pid;
            nptr->bkupftm = bkupftm;
            return (0);
         }
 /*
  * The socket to the intercept code has gone down after the connect but before
  * the adv(name) was successfully completed.
  */
```

```
      return (0);
   }
 /*
  * AT THIS POINT WE KNOW THAT EITHER: (1)  fsock == -1 AND  name == "" (2)
  * fsock >=  0  AND  name != "" IN EITHER CASE WE MUST ALLOCATE A TABLE ENTRY
  */

   if ((nptr = (struct namtable *) malloc(sizeof(struct namtable))) == NULL) {
      perror("NAMTOLIST: MALLOC");
      return (-1);
   }
   bzero((char *) nptr, sizeof(*nptr));
   strcpy(nptr->name, name);
   nptr->status = 'N';
   nptr->lsock = lsock;
   nptr->fsock = fsock;
   nptr->pid = pid;
   nptr->bkupftm = bkupftm;
   nptr->msgs = 0;
   if (namlast == NULL) {
      namfirst = nptr;
      namlast = nptr;
   } else {
      namlast->next = nptr;
      nptr->prev = namlast;
      namlast = nptr;
   }

   return (0);
}
```

```
/*****************************************************************************-
-                                                                            -
-    nline.c                                                                 -
-                                                                            -
-    Purpose - simple utility routine to replace the first newline (0X0A)    -
-              found with a NULL.                                            -
-                                                                            -
-*****************************************************************************/

void            nline(buf, len)
   char            *buf;
   int              len;

{
   char            *charptr;

   for (charptr = buf; charptr < buf + len; ++charptr)
      if (*charptr == 0x0A) {
         *charptr = 0x00;
         break;
```

```
    }
  return;
}
```

```
/*****************************************************************************
-                                                                            -
-    pick_bkup.c                                                             -
-                                                                            -
-    Purpose - called by the FTM when a backup system is required for some  -
-              application which may in the future require restart.          -
-                                                                            -
-****************************************************************************/

#include            "header.h"

int             pick_bkup()
{
   struct namtable *nptr;

   extern struct ftmtable *ftmptr;
   extern int      ftmsize,
                   ftmme;

   extern struct namtable *namfirst;

   int             i,
                   min = 999999,
                   minptr = ftmme,
                  *current;


   current = (int *) malloc(ftmsize * sizeof(*current));
   for (i = 0; i < ftmsize; ++i)
      current[i] = 0;

   for (nptr = namfirst; nptr != NULL; nptr = nptr->next)
      for (i = 0; i < ftmsize; ++i)
         if ((ftmptr + i)->sock == nptr->fsock) {
            ++current[i];
            break;
         }
   for (i = ftmme + 1; i < ftmsize; ++i)
      if ((ftmptr + i)->sock < 0)
         continue;
      else if (current[i] < min) {
         min = current[i];
         minptr = i;
      }
   for (i = 0; i < ftmme; ++i)
      if ((ftmptr + i)->sock < 0)
         continue;
      else if (current[i] < min) {
```

```
        min = current[i];
        minptr = i;
      }
   free((char *) current);
   return (minptr);
}
```

```
/*****************************************************************************
-                                                                            -
-    recv_msg.c                                                              -
-                                                                            -
-    Purpose - reads a message from a socket. Calls recv_pkt to assure the   -
-              reassembly of multiple message packets.                       -
-                                                                            -
-****************************************************************************/

#include        "header.h"

int             recv_msg(s, buf, buflen)
   int             s,
                   buflen;
   char           *buf;

{
   int             len,
                   errx,
                   msglen,
                   bytes_to_go,
                   socknamelen;

   char           *bufnow,
                   this_sock;

   struct msgbuf  *mybuf;

   struct sockaddr sockname;

   if ((mybuf = (struct msgbuf *) malloc(MSGBUF_MAX + PACKETSIZE)) == NULL) {
      perror("SEND_MSG: MALLOC");
      errno = EFAULT;
      return (-1);
   }
   bzero(sockname, sizeof(sockname));
   socknamelen = sizeof(sockname);
   if (getsockname(s, &sockname, &socknamelen) < 0)
      this_sock = 'U';
   else if ((socknamelen > 0) && (sockname.sa_family == AF_INET))
      this_sock = 'I';
   else
      this_sock = 'U';

   bzero((char *) mybuf, MSGBUF_MAX + PACKETSIZE);
```

```
   if ((len = recv_pkt(s, mybuf->cmd, MSGBUF_MIN - MSGBUF_HDR, 'H')) < 0) {
      if (errno == EWOULDBLOCK) {
         free((char *) mybuf);
         errno = EWOULDBLOCK;
         return (-1);
      } else if ((errno == ECONNRESET) || (errno == ENOTCONN)) {
         free((char *) mybuf);
         errno = ENOTCONN;
         return (-1);
      } else {
         errx = errno;
         free((char *) mybuf);
         errno = errx;
         perror("RECV_MSG: RECV_BLK-1 FAILED ON HEADER");
         return (-1);
      }
   }
   if (len == 0) {
      free((char *) mybuf);
      errno = ENOTCONN;
      return (-1);
   }

#if MDEBUG
   printf("DEBUG: RECV_MSG: RECV %d HEADER BYTES ON SOCKET %d, CMD = %s\n",
         len, s, mybuf->cmd);
#endif

   sscanf(mybuf->mlen, "%d", &msglen);
   if (buflen < MSGBUF_MIN - MSGBUF_HDR + msglen) {
      printe("RECV_MSG: BUFFER TOO SHORT\n");
      free((char *) mybuf);
      errno = EFAULT;
      return (-1);
   }
   bufnow = mybuf->msg + 1;
   bytes_to_go = msglen;

#if PACKETSIZE
   if ((this_sock == 'I') && ((bytes_to_go % PACKETSIZE) != 0))
      bytes_to_go = bytes_to_go + PACKETSIZE - (bytes_to_go % PACKETSIZE);
#endif

   if (bytes_to_go > 0) {
      if ((len = recv_pkt(s, bufnow, bytes_to_go, 'B')) < 0)
         if ((errno == EWOULDBLOCK) || (errno == ECONNRESET)
             || (errno == ENOTCONN)) {
            free((char *) mybuf);
            errno = ENOTCONN;
            return (-1);
         } else {
            errx = errno;
            free((char *) mybuf);
            errno = errx;
            perror("RECV_MSG: RECV_BLK-2 FAILED ON MSG");
            return (-1);
         }
      else if (len == 0) {
         free((char *) mybuf);
         errno = ENOTCONN;
         return (-1);
```

```
      } else
         bytes_to_go -= len;
   }

#if MDEBUG
   printf
      ("DEBUG: RECV_MSG: MESSAGE RECEIVED = %s %c %s %s %s %s %s %s %s\n",
       mybuf->cmd,
       mybuf->from,
       mybuf->status,
       mybuf->name_to,
       mybuf->name_from,
       mybuf->seqnum,
       mybuf->errno,
       mybuf->mlen,
       mybuf->msg);
#endif

   bcopy(mybuf->cmd, buf, MSGBUF_MIN - MSGBUF_HDR + msglen);
   free((char *) mybuf);
   return (MSGBUF_MIN - MSGBUF_HDR + msglen);
}
```

```
/*****************************************************************************
-                                                                            -
-    recv_pkt.c                                                              -
-                                                                            -
-    Purpose - reads a message chunk from a socket. This may require the     -
-              reassembly of multiple partial or full packets.               -
-                                                                            -
-****************************************************************************/

#include        "header.h"

int             recv_pkt(s, buf, buflen, type)
   int             s,
                   buflen;
   char           *buf,
                   type;

{
   int             len,
                   wait,
                   bytes_to_go;

   char           *bufnow;

   extern struct ftm_time ftmtime;


   bufnow = buf;
   bytes_to_go = buflen;
```

```
   FD_ZERO(&ftmtime.readfds_temp);
   while (bytes_to_go > 0) {
      wait = 0;
      LOOP {
         if ((len = recv(s, bufnow, bytes_to_go, 0)) > 0) {

#if MDEBUG
            printf("DEBUG: RECV_PKT: RECV %d MSG BYTES ON SOCKET %d\n", len, s);
#endif

            bufnow += len;
            bytes_to_go -= len;
            break;
         } else if (len == 0)
            return (0);
         else if (errno == EWOULDBLOCK)
            if ((type == 'H') && (bytes_to_go == buflen))
               return (-1);
            else if ((wait * MSG2_SLEEP) > (1000000 * MSG2_WAIT)) {
               errno = ENOTCONN;
               return (-1);
            }
         FD_SET(s, &ftmtime.readfds_temp);
         bcopy((char *) &ftmtime.msgtime_perm,
               (char *) &ftmtime.msgtime_temp, sizeof(struct timeval));
         if (select(ftmtime.widthfds, &ftmtime.readfds_temp, NULL, NULL,
                &ftmtime.msgtime_temp) == 0)
            ++wait;
      }
   }
   return (buflen);
}
```

```
/*****************************************************************************-
-                                                                            -
-    rundaemon.c                                                             -
-                                                                            -
-    Purpose - allows the FTMs to be started on remote systems through a     -
-              daemon process so that the FTMs may run in the background     -
-              with no controlling terminal.                                 -
-                                                                            -
-    Note - this routine was written by Ancelin Shah. It and shell script    -
-           rkill are the only code in Appendix B not written by the author  -
-           of this dissertation.                                            -
-                                                                            -
-****************************************************************************/

/*
 * Sessdetach - Detaches a daemon from a log-in session. Compile with a -DBSD
 * switch on Berkeley UNIX systems.
 */
```

```
#include <signal.h>
#include <stdio.h>
#include <sys/param.h>

#ifdef BSD
#include <sys/file.h>
#include <sys/ioctl.h>
#endif

void            sessdetach()
{
   int             fd;

 /* if launched by init (process 1) no need to detach */

   if (getppid() == 1)
      goto out;

 /* ignores terminal stop signals */

#ifdef SIGTTOU
   signal(SIGTTOU, SIG_IGN);
#endif

#ifdef SIGTTIN
   signal(SIGTTIN, SIG_IGN);
#endif


#ifdef SIGTSTP
   signal(SIGTSTP, SIG_IGN);
#endif

   if (fork() != 0) {
      exit(0);
   }
 /* ensure that daemon can't reacquire a new controlling terminal */

#ifdef BSD                      /* on Berkeley UNIX systems */
   setpgrp(0, getpid());        /* change process group leader */
   if ((fd = open("dev/tty", O_RDWR)) >= 0) {
      ioctl(fd, TIOCNOTTY, 0);  /* lose controlling terminal */
      close(fd);
   }
#else                           /* on AT&T unix systems */
   setpgrp();
   if (fork() != 0)
      exit(0);
#endif

out:
   for (fd = 0; fd < NOFILE; fd++)      /* close all open non-terminal files */
      close(fd);
   chdir("/");                  /* change to root directory */
   umask(0);                    /* clear any inherited filemode creation mask */
   return;
}

/*
 * Rundaemon - Launches a daemon program after detaching it from the login
 * session. Usage : rundaemon daemonprog [args] where daemonprog is complete
```

```
 * path name of daemon, rundaemon is program to make daemonprog a daemon, and
 * "args" are 0 or more arguments to pass to daemon.
 */


main(argc, argv)
   int             argc;
   char           *argv[];
{
   if (argc < 2) {
      fprintf(stderr, "usage:rundaemon command\n");
      exit(-1);
   }
   if (access(argv[1], 1) == -1) {
      perror(argv[1]);
      exit(-1);
   }
   sessdetach();
   if (execv(argv[1], &argv[1]) == -1) {        /* start daemon program */
      perror("execv:error :");
      exit(-1);
   }
   exit(-1);                    /* THIS STATEMENT WILL NEVER BE REACHED */
}
```

```
/*****************************************************************************
-                                                                            -
-    send_msg.c                                                              -
-                                                                            -
-    Purpose - sends a message to a socket. This usually requires the        -
-              disassembly header and data message portions.                 -
-                                                                            -
-****************************************************************************/

#include        "header.h"

int             send_msg(s, buf, buflen)
   int             s,
                   buflen;
   char           *buf;

{
   int             len,
                   msglen,
                   msglentot,
                   temp1,
                   temp2,
                   temp3,
                   temp4,
                   temp5,
                   socknamelen;
```

```
char            this_sock;

extern int      int_msgs;

extern char     ftm_ic;

struct msgbuf  *mybuf;

struct namtable *nptr;

struct sockaddr sockname:


if ((mybuf = (struct msgbuf *) malloc(MSGBUF_MAX + PACKETSIZE)) == NULL) {
   perror("SEND_MSG: MALLOC");
   errno = EFAULT;
   return (-1);
}
bzero(sockname, sizeof(sockname));
socknamelen = sizeof(sockname);
if (getsockname(s, &sockname, &socknamelen) < 0)
   this_sock = 'U';
else if ((socknamelen > 0) && (sockname.sa_family == AF_INET))
   this_sock = 'I';
else
   this_sock = 'U';

if (buflen < MSGBUF_MIN - MSGBUF_HDR) {
   printe3("SEND_MSG: LEN = %d,  LESS THAN MIN (HEADER) LENGTH = %d\n",
           buflen, MSGBUF_MIN - MSGBUF_HDR);
   free((char *) mybuf);
   errno = EFAULT;
   return (-1);
}
if (buflen > MSGBUF_MAX - MSGBUF_HDR) {
   printe3("SEND_MSG: LEN = %d,  MORE THAN MAX MESSAGE LENGTH = %d\n",
           buflen, MSGBUF_MAX - MSGBUF_HDR);
   free((char *) mybuf);
   errno = EFAULT;
   return (-1);
}
bzero((char *) mybuf, MSGBUF_MAX + PACKETSIZE);
bcopy(buf, mybuf->cmd, buflen);

if ((len = send(s, mybuf->cmd, MSGBUF_MIN - MSGBUF_HDR, 0)) < 0) {
   if ((errno == EPIPE) || (errno == ENOTCONN)) {
      free((char *) mybuf);
      errno = ENOTCONN;
      return (-1);
   } else {
      perror("SEND_MSG: SEND-1 FAILED ON HEADER");
      free((char *) mybuf);
      errno = EFAULT;
      return (-1);
   }
} else if (len == 0) {
   free((char *) mybuf);
   errno = ENOTCONN;
   return (-1);
} else if (len != MSGBUF_MIN - MSGBUF_HDR) {
   printe3("SEND_MSG: SENT HEADER = %d BYTES BUT TRANSMITTED = %d\n",
```

```
            MSGBUF_MIN - MSGBUF_HDR, len);
      free((char *) mybuf);
      errno = EFAULT;
      return (-1);
   }

#if MDEBUG
   printf("DEBUG: SEND_MSG: SEND %d BYTES TO SOCKET %d, CMD = %s\n",
          len, s, mybuf->cmd);
#endif

   sscanf(mybuf->mlen, "%d", &msglentot);
   if (buflen < MSGBUF_MIN - MSGBUF_HDR + msglentot) {
      printe("SEND_MSG: BUFFER TOO SHORT\n");
      free((char *) mybuf);
      errno = EFAULT;
      return (-1);
   }
   if (msglentot > 0) {
      msglen = msglentot;

#if PACKETSIZE
      if ((this_sock == 'I') && ((msglen % PACKETSIZE) != 0))
         msglen = msglen + PACKETSIZE - (msglen % PACKETSIZE);
#endif

      if ((len = send(s, mybuf->msg + 1, msglen, 0)) == msglen) {

#if MDEBUG
         printf("DEBUG: SEND_MSG: SEND %d BYTES TO SOCKET %d\n", len, s);
#endif

         ;
      } else if (len > 0) {
         printe3("SEND_MSG: SENT MESSAGE = %d BYTES BUT TRANSMITTED = %d\n",
                 msglen, len);
         free((char *) mybuf);
         errno = ENOTCONN;
         return (-1);
      } else if (len == 0) {
         free((char *) mybuf);
         errno = ENOTCONN;
         return (-1);
      } else if ((errno == EPIPE) || (errno == ENOTCONN)) {
         free((char *) mybuf);
         errno = ENOTCONN;
         return (-1);
      } else {
         perror("SEND_MSG: SEND-2");
         printe3("SEND_MSG: SEND-2: ATTEMPT TO SEND %d BYTES TO SOCKET %d\n",
                 msglen, s);
         free((char *) mybuf);
         errno = EFAULT;
         return (-1);
      }

#if MDEBUG
      printf
         ("DEBUG: SEND_MSG: MESSAGE SENT = %s %c %s %s %s %s %s %s %s\n",
          mybuf->cmd,
          mybuf->from,
          mybuf->status,
```

```
            mybuf->name_to,
            mybuf->name_from,
            mybuf->seqnum,
            mybuf->errno,
            mybuf->mlen,
            mybuf->msg);
#endif

   }
/*
   THE FOLLOWING CODE IS EXECUTED ONLY BY THE FTM ON UNIX SOCKETS --------------
*/
   if ((ftm_ic == 'F') && (this_sock == 'U')) {
      temp1 = s;
      temp2 = -1;
      if (namlookup("", &temp1, &temp2, &temp3,
                    &temp4, &temp5, &nptr) >= 0) {

#if FDEBUG
         printf("DEBUG: SEND_MSG: UNIX MSG SENT TO SOCK = %d, NAME = %s\n",
                s, nptr->name);
#endif

         ++nptr->msgs;
         ++int_msgs;
      } else {
         ;

#if FDEBUG
         printf("DEBUG: SEND_MSG: UNIX MSG SENT TO SOCK = %d, ADV FAILED\n", s);
#endif

      }
   }
/*
   ---------------------------------------------------------------------------
*/
   free((char *) mybuf);
   return (MSGBUF_MIN - MSGBUF_HDR + msglentot);
}
```

```
/*****************************************************************************-
-                                                                            -
-    setcap.c                                                                -
-                                                                            -
-    Purpose - utility program to capitalize a character string.             -
-                                                                            -
-*****************************************************************************/

void            setcap(astring, len)
   char           *astring;
   int             len;

{
   char           *ptr;
   int             diff;

   ptr = astring;
   diff = 'A' - 'a';

   for (ptr = astring; ptr < astring + len; ++ptr)
      if ((*ptr >= 'a') && (*ptr <= 'z'))
         *ptr += diff;

   return;
}
```

```
/*****************************************************************************-
-                                                                            -
-    strlcpy.c                                                               -
-                                                                            -
-    Purpose - utility program to copy a string under a maximum length       -
-              restriction.                                                  -
-                                                                            -
-    Use - strlcpy (s1,s2,len)                                               -
-          This causes s2 to be copied to s1, but no more than len           -
-          characters. This means that s2 may not be NULL terminated.        -
-                                                                            -
-*****************************************************************************/

char            *strlcpy(s1, s2, len)
   char           *s1,
                  *s2;
   int             len;

{
   int             i;

   for (i = 0; i < len; ++i) {
      *(s1 + i) = *(s2 + i);
      if (*(s2 + i) == 0x00)
         break;
   }
```

```
   return (s1);
}
```

```
/*****************************************************************************-
-                                                                            -
-    strnlen.c                                                               -
-                                                                            -
-    Purpose - utility program to copy a string under a maximum length       -
-              restriction.                                                  -
-                                                                            -
-    Use - strnlen (s1,len)                                                  -
-          This returns the length of string s1, but stops searching for the -
-          terminating NULL after len characters have been examined.         -
-                                                                            -
-****************************************************************************/

int             strnlen(s1, len)
   char           *s1;
   int             len;

{
   char           *s1ptr;

   for (s1ptr = s1; s1ptr < s1 + len; ++s1ptr)
      if (*s1ptr == 0x00)
         return (s1ptr - s1);

   return (len);
}
```

```
/*****************************************************************************
-                                                                            -
-    to.c                                                                    -
-                                                                            -
-    Purpose - called by application program to send a message to some other -
-              application somewhere in the system.                          -
-                                                                            -
-****************************************************************************/

#include   "header.h"


int             to(name, msg, msglen)
   char           *name,
                  *msg;
   int             msglen;

{
   int             rc,
                   err,
                   oldmask,
                   tox();


   oldmask = sigblock(sigmask(SIGUSR1));
   rc = tox(name, msg, msglen);
   err = errno;
   sigsetmask(oldmask);
   errno = err;
   return (rc);
}


static int      tox(name, msg, msglen)
   char           *name,
                  *msg;
   int             msglen;

{
   char            capname[MAXNAMELEN + 1];

   struct msgbuf  *mptr,
                  *mptr2;

   int             minmsglen;

   extern char     intadvyet,
                   intmyname[MAXNAMELEN + 1];

   extern int      ftmsock,
                   intseqnum,
                   intpidme;

   extern struct ftm_time ftmtime;


   if (strnlen(name, 1) == 0) {
      errno = EINVAL;
      return (-1);
   }
   if ((strnlen(name, 2) == 1) && (name[0] == '*')) {
```

```
        errno = EINVAL;
        return (-1);
    }
    if ((intadvyet != 'Y') || (intpidme != getpid())) {
        printe("TO: ADV NOT YET COMPLETED SUCCESSFULLY\n");
        errno = EFAULT;
        return (-1);
    }
    if (ftmsock < 0) {
        errno = ENOTCONN;
        return (-1);
    }
    minmsglen = msglen;
    if (minmsglen < 0)
        minmsglen = strlen(msg) + 1;
    if (minmsglen > MAXMSGLEN)
        minmsglen = MAXMSGLEN;

    bzero(capname, sizeof(capname));
    strlcpy(capname, name, sizeof(capname) - 1);
    setcap(capname, sizeof(capname) - 1);

    if (((mptr = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL)
        || ((mptr2 = (struct msgbuf *) malloc(MSGBUF_MAX)) == NULL)) {
        perror("TO: MALLOC");
        if (mptr != NULL)
            free((char *) mptr);
        errno = EFAULT;
        return (-1);
    }
    bzero((char *) mptr, MSGBUF_MAX);
    bzero((char *) mptr2, MSGBUF_MAX);

    strcpy(mptr->cmd, "TO");
    mptr->from = 'I';
    sprintf(mptr->status, "%d", -1);
    strcpy(mptr->name_to, capname);
    strcpy(mptr->name_from, intmyname);
    sprintf(mptr->seqnum, "%d", intseqnum);
    sprintf(mptr->errno, "%d", 0);
    sprintf(mptr->mlen, "%d", minmsglen);
    ++intseqnum;
    bcopy(msg, mptr->msg, minmsglen);

    if (send_msg(ftmsock, mptr->cmd,
                MSGBUF_MIN - MSGBUF_HDR + minmsglen) < 0) {
        perror("TO: SEND_MSG");
        free((char *) mptr);
        free((char *) mptr2);
        errno = EFAULT;
        return (-1);
    }

#if IDEBUG
    printf("DEBUG: TO: TO MESSAGE SENT TO LOCAL FTM = %d\n", ftmsock);
    printf("DEBUG: TO: NOW WAITING FOR TOOK\n");
#endif

 /*
  * -   LOOP UNTIL THE TOOK MESSAGE IS RECEIVED
  */
```

```
   LOOP {
      getmsgs(-6);

      if (findbuf("TOOK", "", mptr2) < 0) {
         if (errno != EWOULDBLOCK) {
            printe("TO: BAD RETURN FROM FINDBUF-1\n");
            free((char *) mptr);
            free((char *) mptr2);
            errno = EFAULT;
            return (-1);
         } else;
      } else if (strcmp(mptr2->name_from, capname) != 0) {
         printe2("TO: NAME SENT WAS   %s\n", capname);
         printe2("TO: NAME IN TOOK IS %s\n", mptr2->name_from);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else if (strcmp(mptr->seqnum, mptr2->seqnum) != 0) {
         printe3("TO: TOOK SEQNUM MISMATCH %s %s\n", mptr->seqnum,
                 mptr2->seqnum);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else {

#if IDEBUG
         printf("DEBUG: TO: TOOK MESSAGE RECEIVED\n");
#endif

         free((char *) mptr);
         free((char *) mptr2);
         return (0);
      }

      if (findbuf("TONO", "", mptr2) < 0) {
         if (errno != EWOULDBLOCK) {
            printe("TO: BAD RETURN FROM FINDBUF-1\n");
            free((char *) mptr);
            free((char *) mptr2);
            errno = EFAULT;
            return (-1);
         } else;
      } else if (strcmp(mptr2->name_from, capname) != 0) {
         printe2("TO: NAME SENT WAS   %s\n", capname);
         printe2("TO: NAME IN TONO IS %s\n", mptr2->name_from);
         free((char *) mptr);
         free((char *) mptr2);
         errno = EFAULT;
         return (-1);
      } else if (strcmp(mptr->seqnum, mptr2->seqnum) != 0) {
         printe3("TO: TONO SEQNUM MISMATCH %s %s\n", mptr->seqnum,
                 mptr2->seqnum);
         free((char *) mptr);
         free((char *) mptr2);
         errnc = EFAULT;
         return (-1);
      } else {
         sscanf(mptr2->errno, "%d", &errno);
         free((char *) mptr);
         free((char *) mptr2);
```

```
#if IDEBUG
        printf("DEBUG: TO: TONO MESSAGE RECEIVED\n");
#endif

        return (-1);
     }

     if (ftmsock < 0) {
        errno = ENOTCONN;
        return (-1);
     }
     bcopy((char *) &ftmtime.readfds_perm, (char *) &ftmtime.readfds_temp,
           sizeof(struct fd_set));
     bcopy((char *) &ftmtime.timeval_perm, (char *) &ftmtime.timeval_temp,
           sizeof(struct timeval));
     select(ftmtime.widthfds, &ftmtime.readfds_temp, NULL, NULL,
            &ftmtime.timeval_temp);
   }
}
```

APENDIX C

PERFORMANCE AND VALIDATION TESTING PROGRAMS FOR THE FTM PROTOTYPE

This appendix contains the source code for test programs which validate and compile performance statistics on a running distributed FTM system. The executables for these programs are created during creation of the customized FTM system (see Appendix B). The source code for each test program includes a header block of comments describing the individual test. These source code modules may also be used as simple examples of how to format calls which access the various FTM services, and how to include these calls within an application program.

```
/*****************************************************************************
-                                                                            -
-    zadvclose.c                                                             -
-                                                                            -
-    Purpose - a small test program to time the delay in "adv" and "ftmclose" -
-              calls.                                                        -
-                                                                            -
-    Call - zadvclose [loops]                                                -
-                                                                            -
-    Defaults                                                                -
-        loops         100                                                   -
-                                                                            -
-****************************************************************************/

#include     "header.h"

#define       DEF_LOOPS   100

int             main(argc, argv)
   int             argc;
   char           **argv;

{
   char            name[21];

   int             i,
                   loops = DEF_LOOPS;

   struct timeb    time1,
                   time2;

   double          maxa = 0.0,
                   maxb = 0.0,
                   mina = 2000000000.0,
                   minb = 2000000000.0,
                   delay,
                   avg,
                   suma = 0.0,
                   sumb = 0.0,
                   sumsqrsa = 0.0,
                   sumsqrsb = 0.0,
                   stdev;


   if (argc > 1)
      sscanf(argv[1], "%d", &loops);
   if (loops < 1)
      loops = 1;
   printf("ZADVCLOSE: NUMBER OF LOOPS = %d\n", loops);

   for (i = 1; i <= loops; ++i) {
      ftime(&time1);
      if (adv("My_Name_is_advclose") < 0) {
         printf("ZADVCLOSE: ADV BAD RETURN, ERRNO = %d, LOOP = %d\n", errno, i);
         exit(-1);
      }
      ftime(&time2);
      delay = time2.time - time1.time
         + ((short) time2.millitm - (short) time1.millitm) / 1000.0;
      if (delay > maxa)
         maxa = delay;
      if (delay < mina)
```

```
            mina = delay;
         suma += delay;
         sumsqrsa += delay * delay;

         ftime(&time1);
         if (ftmclose() < 0) {
            printf("ZADVCLOSE: FTMCLOSE BAD RETURN, ERRNO = %d, LOOP = %d\n",
                   errno, i);
            exit(-1);
         }
         ftime(&time2);
         delay = time2.time - time1.time
            + ((short) time2.millitm - (short) time1.millitm) / 1000.0;
         if (delay > maxb)
            maxb = delay;
         if (delay < minb)
            minb = delay;
         sumb += delay;
         sumsqrsb += delay * delay;
         if ((i % 100) == 0)
            printf("LOOPS = %d\n", i);
      }
   /*
   -    calculate the statistics
   */
      avg = suma / loops;
      if (loops > 1)
         stdev = sqrt((sumsqrsa - (suma * suma / loops)) / (loops - 1));
      else
         stdev = 0.0;
      printf("ADV: MAX = %6.3f,   MIN = %6.3f,   AVG = %6.3f,   STDEV = %6.3f\n",
             maxa, mina, avg, stdev);

      avg = sumb / loops;
      if (loops > 1)
         stdev = sqrt((sumsqrsb - (sumb * sumb / loops)) / (loops - 1));
      else
         stdev = 0.0;
      printf("CLS: MAX = %6.3f,   MIN = %6.3f,   AVG = %6.3f,   STDEV = %6.3f\n",
             maxb, minb, avg, stdev);
      exit(0);
   }
```

```
/*****************************************************************************
-                                                                            -
-    zdummy.c                                                                -
-                                                                            -
-    Purpose - advertises "dummy" and then reads (and discards) any incoming -
-              messages. This allows the sending program (in this case "zto") -
-              to generate timing statistics.                                -
-                                                                            -
-****************************************************************************/

#include     "header.h"

#define       MAX_SIZE  4096

main()
{
   char            name[MAXNAMELEN + 1],
                   buf[MAX_SIZE];


   printf("ZDUMMY: ABOUT TO ADV\n");

   if (adv("dummy") < 0) {
      printf("ZDUMMY: ADV BAD RETURN\n");
      return (-1);
   }
   printf("ZDUMMY: ADVERTISE WAS SUCCESSFUL\n");

   LOOP {
      strcpy(name, "*");
      while (from(name, buf, MAX_SIZE) >= 0)
         strcpy(name, "*");
      if (errno != EWOULDBLOCK)
         exit(0);
      pause();
   }
}
```

```
/*****************************************************************************
-                                                                           -
-    zflop.c                                                                -
-                                                                           -
-    Call - zflop [loops] [n]                                               -
-                                                                           -
-    Purpose - a small test program which "flops" around the n node FTM     -
-              system loops times. It has a critical file (which contains the -
-              initial start time and a count of restarts). After loops     -
-              restarts, the program reports the timing information (in the -
-              final critical file) and then terminates.                    -
-                                                                           -
-    Note - since the clocks are distributed, accurate timing requires that -
-           the timing statistics be cumulated on a single system. This     -
-           program assumes that it is being run on a n node distributed FTM -
-           system with no other connected applications so that it will     -
-           "flop" around the system in a constant pattern. If n is 2 then  -
-           this program will "flop" back and forth regardless of additional -
-           connected applications although they, along with other system   -
-           workload, will bias the timing statistics. Timing statistics will -
-           be taken every other n flops, and divided by n. If not a multiple -
-           of n, then the parameter loops will be incremented to the next  -
-           multiple.                                                       -
-                                                                           -
-    Defaults:                                                              -
-           loops   100                                                     -
-           n         2                                                     -
-                                                                           -
-     Important - make sure that there are no pre-existing critical files   -
-           (#define CFILE) on any of the n nodes or this application will not -
-           know that it is just starting out.                              -
-                                                                           -
-****************************************************************************/

#include     <stdio.h>
#include     <errno.h>
#include     <string.h>
#include     <math.h>
#include     <sys/types.h>
#include     <sys/timeb.h>

#define       DEF_LOOPS   100
#define       DEF_N         2
#define       CFILE     "/tmp/auvschi/ftm/zflop.crt"
#define       PFILE     "/u/auvusers1/auvschi/ftm/zflop.lis"
#define       DEBUG         0

int             main(argc, argv)
   int             argc;
   char          **argv;

{
   int             i,
                   n = DEF_N,
                   loops = DEF_LOOPS;

   struct timeb    time0,
                   time1,
                   time2;

   double          sum = 0.0,
                   sumsqrs = 0.0,
```

```
                    avg,
                    stdev,
                    delay,
                    max = 0.0,
                    min = 2000000000.0;

   FILE            *critical,
                   *pfile;


#if DEBUG
   pfile = fopen(PFILE, "a");
#endif

   if (argc > 1)
      sscanf(argv[1], "%d", &loops);
   if (argc > 2)
      sscanf(argv[2], "%d", &n);
   if ((loops % n) != 0)
      loops += ((loops / n) + 1) * n - loops;

#if DEBUG
   fprintf(pfile, "ZFLOP: NUMBER OF LOOPS = %d\n", loops);
   fprintf(pfile, "ZFLOP: NUMBER OF NODES = %d\n", n);
   fflush(pfile);

   fprintf(pfile, "ZFLOP: ABOUT TO ADV\n");
   fflush(pfile);
#endif

   if (adv("Flop") < 0) {

#if DEBUG
      fprintf(pfile, "ZFLOP: ADV BAD RETURN\n");
      fflush(pfile);
#endif

      exit(-1);
   }

#if DEBUG
   fprintf(pfile, "ZFLOP: ABOUT TO OPEN FILE\n");
   fflush(pfile);
#endif

   if ((critical = fopen(CFILE, "r")) != NULL) {

#if DEBUG
      fprintf(pfile, "ZFLOP: HAVE OPENED CRITICAL FILE\n");
      fflush(pfile);
#endif

      fread(&sum, 1, sizeof(sum), critical);
      fread(&sumsqrs, 1, sizeof(sumsqrs), critical);
      fread(&max, 1, sizeof(max), critical);
      fread(&min, 1, sizeof(min), critical);
      fscanf(critical, "%d %hd %d %hd %d %d %d", &time0.time, &time0.millitm,
             &time1.time, &time1.millitm, &loops, &n, &i);
      ++i;
      fclose(critical);

#if DEBUG
```

```
      fprintf(pfile, "ZFLOP: CRITICAL FILE READ, LOOPS = %d, N = %d, I = %d\n",
              loops, n, i);
      fflush(pfile);
#endif
   } else {

#if DEBUG
      fprintf(pfile, "ZFLOP: NO CRITICAL FILE - NOT YET RESTARTED\n");
      fflush(pfile);
#endif

      ftime(&time0);

#if DEBUG
      fprintf(pfile, "ZFLOP: TIME0 = %d %d\n", time0.time, time0.millitm);
      fflush(pfile);
#endif

      time1.time = time0.time;
      time1.millitm = time0.millitm;
      i = 0;
   }
   if ((i != 0) && ((i % n) == 0)) {    /* EVEN LOOP - CALCULATF STATISTICS   */
      ftime(&time2);

#if DEBUG
      fprintf(pfile, "ZFLOP: TIME2 = %d %d\n", time2.time, time2.millitm);
      fflush(pfile);
#endif

      delay = (time2.time - time1.time)
         + ((short) time2.millitm - (short) time1.millitm) / 1000.0;
      sum += delay;
      delay /= n;
      sumsqrs += pow(delay, 2.0) * n;

#if DEBUG
      fprintf(pfile, "ZFLOP: LOOP = %d,   SUM = %f,   sumsqrs = %f\n",
              i, sum, sumsqrs);
      fflush(pfile);
#endif

      if (delay > max)
         max = delay;
      if (delay < min)
         min = delay;
      time1.time = time2.time;
      time1.millitm = time2.millitm;

#if DEBUG
      fprintf(pfile, "ZFLOP: EVEN NON-ZERO LOOP, LOOPS = %d, N = %d, I = %d\n",
              loops, n, i);
      fflush(pfile);
#endif
   }
   if ((critical = fopen(CFILE, "w")) == NULL) {
      ftmclose();

#if DEBUG
      fprintf(pfile, "ZFLOP: CRITICAL FILE OPEN ERROR\n");
      fflush(pfile);
#endif
```

```
      exit(-1);                    /* FORGET IT   */
   }
   if (i < loops) {
      fwrite(&sum, 1, sizeof(sum), critical);
      fwrite(&sumsqrs, 1, sizeof(sumsqrs), critical);
      fwrite(&max, 1, sizeof(max), critical);
      fwrite(&min, 1, sizeof(min), critical);
      fprin f(critical, "%d %d %d %d %d %d %d\n", time0.time, time0.millitm,
             time1.time, time1.millitm, loops, n, i);
      fclose(critical);
      if (copycf("flop", CFILE) < 0) {

#if DEBUG
         fprintf(pfile, "ZFLOP: COPYCF FAILED FOR ERRNO = %d\n", errno);
         fclose(pfile);
#endif

         ftmclose();
         exit(-1);
      }

#if DEBUG
      fprintf(pfile, "ZFLOP: CRITICAL FILE COPIED, LOOPS = %d, I = %d\n", loops,
 i);
      fclose(pfile);
#endif

      exit(-1);                    /* TERMINATE PREMATURELY   */
   }
/*
   HERE WE HAVE FLOPPED OUR LIMIT
*/

#if DEBUG
   fprintf(pfile, "ZFLOP: TIMES = %d %d %d %d\n",
           time2.time, time2.millitm, time0.time, time0.millitm);
   fflush(pfile);
#endif

   delay = (time2.time - time0.time)
      + ((short) time2.millitm - (short) time0.millitm) / 1000.0;
   fprintf(critical, "%d FLOPS TOOK %7.3f SECONDS, AVERAGE = %5.3f\n",
           i, delay, delay / i);
   avg = sum / loops;
   stdev = sqrt((sumsqrs - (sum * sum / loops)) / (loops - 1));
   fprintf(critical,
           "MAX = %6.3f,   MIN = %6.3f,   AVG = %6.3f,   STDEV = %6.3f\n",
           max, min, avg, stdev);
   fclose(critical);
   ftmclose();                     /* NOW WE CAN SAFELY TERMINATE   */

#if DEBUG
   fprintf(pfile, "ZFLOP FINISHED, LOOPS = %d, I = %d\n", loops, i);
   fclose(pfile);
#endif

   exit(0);
}
```

```
/*****************************************************************************
-                                                                            -
-    zstatus.c                                                               -
-                                                                            -
-    Purpose - a small test program to time the delay in a "ftmstatus" call  -
-                                                                            -
-    Call - zstatus [loops]                                                  -
-                                                                            -
-    Defaults                                                                -
-        loops         100                                                   -
-                                                                            -
-****************************************************************************/

#include     "header.h"

#define       DEF_LOOPS   100

int             main(argc, argv)
   int             argc;
   char          **argv;

{
   char            name[MAXNAMELEN + 1],
                   currsys[MAXHOSTNAMELEN + 1],
                   bkupsys[MAXHOSTNAMELEN + 1];

   int             i,
                   errnox,
                   loops = DEF_LOOPS,
                   mirror,
                   status;

   struct timeb    time1,
                   time2;

   double          max = 0.0,
                   min = 2000000000.0,
                   delay,
                   avg,
                   sum = 0.0,
                   sumsqrs = 0.0,
                   stdev;


   if (argc > 1)
      sscanf(argv[1], "%d", &loops);
   if (loops < 1)
      loops = 1;
   printf("ZSTATUS: NUMBER OF LOOPS = %d\n", loops);

   printf("ZSTATUS: ABOUT TO ADV\n");

   if (adv("My_Name_is_status") < 0) {
      printf("ZSTATUS: ADV BAD RETURN\n");
      return (-1);
   }
   printf("ZSTATUS: ABOUT TO FTMSTATUS\n");

/*
-    gather the statistics
*/
   ftime(&time1);
```

```
   for (i = 0; i < loops; ++i) {
      if (ftmstatus(2, &status, &mirror) < 0) {
         errnox = errno;
         printf("ZSTATUS: FTMSTATUS - BAD RETURN AFTER START\n");
         printf("ZSTATUS: FTMSTATUS - errno = %d\n", errnox);
         printf("ZSTATUS: FTMSTATUS - LOOP INDEX = %d\n", i);
         exit(-1);
      }
      ftime(&time2);
      delay = time2.time - time1.time
         + ((short) time2.millitm - (short) time1.millitm) / 1000.0;
      if (delay > max)
         max = delay;
      if (delay < min)
         min = delay;
      sum += delay;
      sumsqrs += delay * delay;
      time1.time = time2.time;
      time1.millitm = time2.millitm;
   }
   avg = sum / loops;
   if (loops > 1)
      stdev = sqrt((sumsqrs - (sum * sum / loops)) / (loops - 1));
   else
      stdev = 0.0;
   printf("MAX = %6.3f,   MIN = %6.3f,   AVG = %6.3f,   STDEV = %6.3f\n",
         max, min, avg, stdev);
   exit(0);
}
```

```
/*****************************************************************************
-                                                                            -
-    ztest2.c                                                                -
-                                                                            -
-    Purpose - a small test program to demonstrate how to:                   -
-              (1) advertise a name                                          -
-              (2) send some messages                                        -
-              (3) receive some messages                                     -
-                                                                            -
-                                                                            -
-    Note - this test program works along with ztest4.c, ztestc.c,           -
-           and ztestd.c. The four programs must be running simultaneously.  -
-           Each may be running on any system with a connected FTM.          -
-                                                                            -
-****************************************************************************/

#include     <stdio.h>
#include     <errno.h>
#include     <string.h>

main()
{
```

```
char            msg[100],
                name[21];

int             i;


printf("TEST2: ABOUT TO ADV\n");

if (adv("boing") < 0) {
   printf("TEST2: ADV BAD RETURN\n");
   return (-1);
}
printf("TEST2: ABOUT TO TO\n");

while (to("eeps", "Hi eeps from boing", -1) < 0)
   if (errno != EWOULDBLOCK) {
      printf("TEST2: TO - BAD RETURN\n");
      return (-1);
   } else {
      printf("TEST2: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
      sleep(2);
   }

while (to("gleep", "Hi gleep from boing", -1) < 0)
   if (errno != EWOULDBLOCK) {
      printf("TEST2: TO - BAD RETURN\n");
      return (-1);
   } else {
      printf("TEST2: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
      sleep(2);
   }

while (to("groan", "Hi groan from boing", -1) < 0)
   if (errno != EWOULDBLOCK) {
      printf("TEST2: TO - BAD RETURN\n");
      return (-1);
   } else {
      printf("TEST2: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
      sleep(2);
   }

printf("TEST2: ABOUT TO LOOP FOR FROM\n");

while (from("groan", msg, 100) < 0)
   if (errno != EWOULDBLOCK) {
      printf("TEST2: FROM - BAD RETURN\n");
      return (-1);
   } else {
      printf("TEST2: FROM REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
      sleep(2);
   }
printf("TEST2: THE MESSAGE IS: %s\n", msg);

for (i = 1; i <= 2; ++i) {
   strcpy(name, "*");
   while (from(name, msg, 100) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TEST2: FROM - BAD RETURN\n");
         return (-1);
      } else {
         printf("TEST2: FROM REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
```

```
        }
      printf("TEST2: MESSAGE RECEIVED FROM: %s\n", name);
      printf("TEST2: THE MESSAGE IS: %s\n", msg);
   }

   return (0);
}
```

```
/*****************************************************************************
-                                                                           -
-    ztest4.c                                                               -
-                                                                           -
-    Purpose - a small test program to demonstrate how to:                  -
-              (1) advertise a name                                         -
-              (2) send some messages                                       -
-              (3) receive some messages                                    -
-                                                                           -
-                                                                           -
-    Note - this test program works along with ztest2.c, ztestc.c,          -
-           and ztestd.c. The four programs must be running simultaneously. -
-           Each may be running on any system with a connected FTM.         -
-                                                                           -
-****************************************************************************/

#include     <stdio.h>
#include     <errno.h>
#include     <string.h>

main()
{
   char            msg[100],
                   name[21];

   int             i;


   printf("TEST4: ABOUT TO ADV\n");

   if (adv("eeps") < 0) {
      printf("TEST4: ADV BAD RETURN\n");
      return (-1);
   }
   printf("TEST4: ABOUT TO TO\n");

   while (to("gleep", "Hi gleep from eeps", -1) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TEST4: TO - BAD RETURN\n");
         return (-1);
      } else {
         printf("TEST4: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
      }
```

```
    while (to("groan", "Hi groan from eeps", -1) < 0)
       if (errno != EWOULDBLOCK) {
          printf("TEST4: TO - BAD RETURN\n");
          return (-1);
       } else {
          printf("TEST4: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
          sleep(2);
       }

    while (to("boing", "Hi boing from eeps", -1) < 0)
       if (errno != EWOULDBLOCK) {
          printf("TEST4: TO - BAD RETURN\n");
          return (-1);
       } else {
          printf("TEST4: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
          sleep(2);
       }

    printf("TEST4: ABOUT TO LOOP FOR FROM\n");

    while (from("boing", msg, 100) < 0)
       if (errno != EWOULDBLOCK) {
          printf("TEST4: FROM - BAD RETURN\n");
          return (-1);
       } else {
          printf("TEST4: FROM REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
          sleep(2);
       }
    printf("TEST4: THE MESSAGE IS: %s\n", msg);

    for (i = 1; i <= 2; ++i) {
       strcpy(name, "*");
       while (from(name, msg, 100) < 0)
          if (errno != EWOULDBLOCK) {
             printf("TEST4: FROM - BAD RETURN\n");
             return (-1);
          } else {
             printf("TEST4: FROM REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
             sleep(2);
          }
       printf("TEST4: MESSAGE RECEIVED FROM: %s\n", name);
       printf("TEST4: THE MESSAGE IS: %s\n", msg);
    }

    return (0);
}
```

```
/*****************************************************************************-
-                                                                            -
-    ztestc.c                                                                -
-                                                                            -
-    Purpose - a small test program to demonstrate how to:                   -
-              (1) advertise a name                                          -
-              (2) send some messages                                        -
-              (3) receive some messages                                     -
-                                                                            -
-                                                                            -
-    Note - this test program works along with ztest2.c, ztest4.c,           -
-           and ztestd.c. The four programs must be running simultaneously.  -
-           Each may be running on any system with a connected FTM.          -
-                                                                            -
-*****************************************************************************/

#include     <stdio.h>
#include     <errno.h>
#include     <string.h>

main()
{
   char            msg[100],
                   name[21];

   int             i;


   printf("TESTc: ABOUT TO ADV\n");

   if (adv("gleep") < 0) {
      printf("TESTc: ADV BAD RETURN\n");
      return (-1);
   }
   printf("TESTc: ABOUT TO TO\n");

   while (to("groan", "Hi groan from gleep", -1) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TESTc: TO - BAD RETURN\n");
         return (-1);
      } else {
         printf("TESTc: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
      }


   while (to("boing", "Hi boing from gleep", -1) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TESTc: TO - BAD RETURN\n");
         return (-1);
      } else {
         printf("TESTc: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
      }

   while (to("eeps", "Hi eeps from gleep", -1) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TESTc: TO - BAD RETURN\n");
         return (-1);
      } else {
         printf("TESTc: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
```

```
    }

  printf("TESTc: ABOUT TO LOOP FOR FROM\n");

  while (from("eeps", msg, 100) < 0)
     if (errno != EWOULDBLOCK) {
        printf("TESTc: FROM - BAD RETURN\n");
        return (-1);
     } else {
        printf("TESTc: FROM REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
        sleep(2);
     }
  printf("TESTc: THE MESSAGE IS: %s\n", msg);

  for (i = 1; i <= 2; ++i) {
     strcpy(name, "*");
     while (from(name, msg, 100) < 0)
        if (errno != EWOULDBLOCK) {
           printf("TESTc: FROM - BAD RETURN\n");
           return (-1);
        } else {
           printf("TESTc: FROM REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
           sleep(2);
        }
     printf("TESTc: MESSAGE RECEIVED FROM: %s\n", name);
     printf("TESTc: THE MESSAGE IS: %s\n", msg);
  }

  return (0);
}
```

```
/*****************************************************************************-
-                                                                            -
-    ztestd.c                                                                -
-                                                                            -
-    Purpose - a small test program to demonstrate how to:                   -
-              (1) advertise a name                                          -
-              (2) send some messages                                        -
-              (3) receive some messages                                     -
-                                                                            -
-                                                                            -
-    Note - this test program works along with ztest2.c, ztest4.c,           -
-           and ztestc.c. The four programs must be running simultaneously.  -
-           Each may be running on any system with a connected FTM.          -
-                                                                            -
-*****************************************************************************/

#include     <stdio.h>
#include     <errno.h>
#include     <string.h>

main()
```

```
{
   char            msg[100],
                   name[21];

   int             i;


   printf("TESTd: ABOUT TO ADV\n");

   if (adv("groan") < 0) {
      printf("TESTd: ADV BAD RETURN\n");
      return (-1);
   }
   printf("TESTd: ABOUT TO TO\n");

   while (to("boing", "Hi boing from groan", -1) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TESTd: TO - BAD RETURN\n");
         return (-1);
      } else {
         printf("TESTd: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
      }

   while (to("eeps", "Hi eeps from groan", -1) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TESTd: TO - BAD RETURN\n");
         return (-1);
      } else {
         printf("TESTd: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
      }

   while (to("gleep", "Hi gleep from groan", -1) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TESTd: TO - BAD RETURN\n");
         return (-1);
      } else {
         printf("TESTd: TO REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
      }

   printf("TESTd: ABOUT TO LOOP FOR FROM\n");

   while (from("gleep", msg, 100) < 0)
      if (errno != EWOULDBLOCK) {
         printf("TESTd: FROM - BAD RETURN\n");
         return (-1);
      } else {
         printf("TESTd: FROM REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
         sleep(2);
      }
   printf("TESTd: THE MESSAGE IS: %s\n", msg);

   for (i = 1; i <= 2; ++i) {
      strcpy(name, "*");
      while (from(name, msg, 100) < 0)
         if (errno != EWOULDBLOCK) {
            printf("TESTd: FROM - BAD RETURN\n");
            return (-1);
         } else {
            printf("TESTd: FROM REJECTED - WILL TRY AGAIN IN 2 SECONDS\n");
```

```
            sleep(2);
         }
      printf("TESTd: MESSAGE RECEIVED FROM: %s\n", name);
      printf("TESTd: THE MESSAGE IS: %s/n", msg);
   }

   return (0);
}
```

```
/*****************************************************************************
-                                                                            -
-    zto.c                                                                   -
-                                                                            -
-    Purpose - a small test program to time the delay in a "to" call         -
-                                                                            -
-    Call - zto [loops] [message_size]                                       -
-                                                                            -
-    Defaults                                                                -
-        loops         100                                                   -
-        message_size  400                                                   -
-                                                                            -
-    Note - requires program "zdummy" to serve as message target.            -
-                                                                            -
-****************************************************************************/

#include     "header.h"

#define       MAX_SIZE   4096
#define       DEF_SIZE    400
#define       DEF_LOOPS   100

int             main(argc, argv)
   int             argc;
   char          **argv;

{
   char            msg[MAX_SIZE],
                   name[21];

   int             i,
                   errnox,
                   size = DEF_SIZE,
                   loops = DEF_LOOPS;

   struct timeb    time1,
                   time2;

   double          max = 0.0,
                   min = 2000000000.0,
                   delay,
                   avg,
                   sum = 0.0,
```

```
                    sumsqrs = 0.0,
                    stdev;

   if (argc > 1)
      sscanf(argv[1], "%d", &loops);
   if (loops < 1)
      loops = 1;
   printf("ZTO: NUMBER OF LOOPS = %d\n", loops);
   if (argc > 2)
      sscanf(argv[2], "%d", &size);
   if (size < 0)
      size = 0;
   printf("ZTO: MESSAGE SIZE = %d\n", size);
   if (size > MAX_SIZE) {
      printf("MESSAGE SIZE TOO BIG - LIMIT IS %d\n", MAX_SIZE);
      exit(-1);
   }
   if (size > 0) {
      msg[0] = 'Y';
      if (size > 2)
         msg[size - 2] = 'Y';
      for (i = 1; i < size - 2; ++i)
         msg[i] = 'X';
      msg[size - 1] = 0x00;
   }
   printf("ZTO: ABOUT TO ADV\n");

   if (adv("My_Name_is_to") < 0) {
      printf("ZTO: ADV BAD RETURN\n");
      return (-1);
   }
   printf("ZTO: ABOUT TO TO\n");

/*
-    make sure that zdummy is active
*/
   if (to("dummy", msg, size) < 0) {
      if (errno != EWOULDBLOCK) {
         printf("ZTO: TO - BAD RETURN\n");
         exit(-1);
      } else {
         printf("ZTO: TO - zdummy NOT ACTIVE\n");
         exit(-1);
      }
   } else
      printf("ZTO: INITIAL TO SUCCESSFUL\n");
/*
-    gather the statistics
*/
   ftime(&time1);
   for (i = 0; i < loops; ++i) {
      if (to("dummy", msg, size) < 0) {
         errnox = errno;
         printf("ZTO: TO - BAD RETURN AFTER START\n");
         printf("ZTO: TO - errno = %d\n", errnox);
         printf("ZTO: TO - LOOP INDEX = %d\n", i);
         exit(-1);
      }
      ftime(&time2);
      delay = time2.time - time1.time
         + ((short) time2.millitm - (short) time1.millitm) / 1000.0;
```

```
      if (delay > max)
         max = delay;
      if (delay < min)
         min = delay;
      sum += delay;
      sumsqrs += delay * delay;
      time1.time = time2.time;
      time1.millitm = time2.millitm;
   }
   avg = sum / loops;
   if (loops > 1)
      stdev = sqrt((sumsqrs - (sum * sum / loops)) / (loops - 1));
   else
      stdev = 0.0;
   printf("MAX = %6.3f,   MIN = %6.3f,   AVG = %6.3f,   STDEV = %6.3f\n",
         max, min, avg, stdev);
   exit(0);
}
```

```
/*****************************************************************************-
-                                                                            -
-    zwhere.c                                                                -
-                                                                            -
-    Purpose - a small test program to time the delay in a "ftmwhere" call   -
-                                                                            -
-    Call - zwhere [loops]                                                   -
-                                                                            -
-    Defaults                                                                -
-        loops         100                                                   -
-                                                                            -
-    Note - requires program "zdummy" to serve as locator target.            -
-                                                                            -
-*****************************************************************************/

#include     "header.h"

#define       DEF_LOOPS   100

int             main(argc, argv)
   int             argc;
   char          **argv;

{
   char            name[MAXNAMELEN + 1],
                   currsys[MAXHOSTNAMELEN + 1],
                   bkupsys[MAXHOSTNAMELEN + 1];

   int             i,
                   errnox,
                   loops = DEF_LOOPS;

   struct timeb    time1,
```

```
                time2;

double          max = 0.0,
                min = 2000000000.0,
                delay,
                avg,
                sum = 0.0,
                sumsqrs = 0.0,
                stdev;


if (argc > 1)
   sscanf(argv[1], "%d", &loops);
if (loops < 1)
   loops = 1;
printf("ZWHERE: NUMBER OF LOOPS = %d\n", loops);

printf("ZWHERE: ABOUT TO ADV\n");

if (adv("My_Name_is_where") < 0) {
   printf("ZWHERE: ADV BAD RETURN\n");
   return (-1);
}
printf("ZWHERE: ABOUT TO FTMWHERE\n");

/*
-    make sure that zdummy is active
*/
if (ftmwhere("dummy", currsys, bkupsys) < 0) {
   printf("ZWHERE: FTMWHERE - BAD RETURN\n");
   exit(-1);
} else if (currsys[0] == 0x00) {
   printf("ZWHERE: FTMWHERE - zdummy NOT ACTIVE\n");
   exit(-1);
} else
   printf("ZWHERE: INITIAL WHERE SUCCESSFUL\n");
sleep(1);
/*
-    gather the statistics
*/
ftime(&time1);
for (i = 0; i < loops; ++i) {
   if ((ftmwhere("dummy", currsys, bkupsys) < 0) || (currsys[0] == 0x00)) {
      errnox = 0;
      if (currsys[0] == 0x00)
         errnox = errno;
      printf("ZWHERE: FTMWHERE - BAD RETURN AFTER START\n");
      printf("ZWHERE: FTMWHERE - errno = %d\n", errnox);
      printf("ZWHERE: FTMWHERE - LOOP INDEX = %d\n", i);
      exit(-1);
   }
   ftime(&time2);
   delay = time2.time - time1.time
      + ((short) time2.millitm - (short) time1.millitm) / 1000.0;
   if (delay > max)
      max = delay;
   if (delay < min)
      min = delay;
   sum += delay;
   sumsqrs += delay * delay;
   time1.time = time2.time;
   time1.millitm = time2.millitm;
```

```
   }
   avg = sum / loops;
   if (loops > 1)
      stdev = sqrt((sumsqrs - (sum * sum / loops)) / (loops - 1));
   else
      stdev = 0.0;
   printf("MAX = %6.3f,   MIN = %6.3f,   AVG = %6.3f,   STDEV = %6.3f\n",
         max, min, avg, stdev);
   exit(0);
}
```

```
/*****************************************************************************-
-                                                                            -
-    zzdummy_in.c                                                            -
-                                                                            -
-    Purpose - receives messages over a socket connected to program zzto_in. -
-                                                                            -
-    NOTE - this program supports INET sockets directly. It does not require -
-           support from an FTM system.                                      -
-                                                                            -
-****************************************************************************/

#include          <stdio.h>
#include          <sys/types.h>
#include          <sys/socket.h>
#include          <netinet/in.h>

#define            FTMPORT   9738
#define            MAX_SIZE  4096


int             main()
{
   struct sockaddr_in server,
                      client;

   int                s1,
                      s2,
                      len,
                      rc,
                      size,
                      bytes_so_far;

   char               buf[MAX_SIZE];

   if ((s1 = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("ZZDUMMY_IN: SOCKET");
      exit(-1);
   }
   bzero((char *) &server, sizeof(server));
   server.sin_port = FTMPORT;
   if (bind(s1, &server, sizeof(server)) < 0) {
```

```
        perror("ZZDUMMY_IN: BIND");
        printf("ZZDUMMY_IN: BIND ADDR = %d \n", FTMPORT);
        exit(-1);
    }
    if (listen(s1, 1) < 0) {
        perror("ZZDUMMY_IN: LISTEN");
        exit(-1);
    }
    if ((s2 = accept(s1, &client, &len)) < 0) {
        perror("ZZDUMMY_IN: ACCEPT");
        exit(-1);
    }
    if (close(s1) < 0) {
        perror("ZZDUMMY_IN: CLOSE LISTENER SOCKET\n");
        exit(-1);
    }
    if ((rc = recv(s2, &size, sizeof(size), 0)) != sizeof(size)) {
        perror("ZZDUMMY_IN: INITIAL RECV");
        exit(-1);
    } else if (rc == 0) {
        printf("ZZDUMMY_IN: IR: RC = 0 => UNIX SOCKET IS DOWN\n");
        exit(-1);
    }
    if (send(s2, "Z", 1) <= 0) {
        perror("ZZDUMMY_IN: INITIAL SEND");
        exit(-1);
    }
    if (size > MAX_SIZE) {
        printf("ZZDUMMY_IN: SIZE = %d LARGER THAN MAX_SIZE = %d\n",
                size, MAX_SIZE);
        exit(-1);
    }
    bzero(buf, size);
    bytes_so_far = 0;
    for (;;) {
        if ((rc = recv(s2, buf, size, 0)) < 0) {
            perror("ZZDUMMY_IN: SUBSEQUENT RECV");
            exit(-1);
        } else if (rc == 0) {
            printf("ZZDUMMY_IN: SR: RC = 0 => INET SOCKET IS DOWN\n");
            exit(-1);
        }
        bytes_so_far += rc;
        if ((bytes_so_far / size) >= 1)
            if (send(s2, "Z", 1) <= 0) {
                perror("ZZDUMMY_IN: SUBSEQUENT SEND");
                exit(-1);
            }
        bytes_so_far %= size;
    }
}
```

```
/*****************************************************************************
-                                                                            -
-    zzdummy_un.c                                                            -
-                                                                            -
-    Purpose - receives messages over a socket connected to program zzto_un. -
-                                                                            -
-    NOTE - this program supports UNIX sockets directly. It does not require -
-           support from an FTM system.                                      -
-                                                                            -
-****************************************************************************/

#include          <stdio.h>
#include          <sys/types.h>
#include          <sys/socket.h>
#include          <sys/un.h>

#define            FTMFAM   "/tmp/auvscni/zzmysocket.sck"
#define            MAX_SIZE  4096


int             main()
{
   struct sockaddr_un server,
                      client;

   int                s1,
                      s2,
                      rc,
                      len,
                      size,
                      bytes_so_far;

   char               buf[MAX_SIZE];


   if ((s1 = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
      perror("ZZDUMMY_UN: SOCKET");
      exit(-1);
   }
   bzero((char *) &server, sizeof(server));
   server.sun_family = AF_UNIX;
   strcpy(server.sun_path, FTMFAM);
   unlink(FTMFAM);
   if (bind(s1, &server, strlen(FTMFAM) + 2) < 0) {
      perror("ZZDUMMY_UN: BIND");
      printf("ZZDUMMY_UN: BIND ADDR = %s \n", FTMFAM);
      exit(-1);
   }
   if (listen(s1, 1) < 0) {
      perror("ZZDUMMY_UN: LISTEN");
      exit(-1);
   }
   if ((s2 = accept(s1, &client, &len)) < 0) {
      perror("ZZDUMMY_UN: ACCEPT");
      exit(-1);
   }
   if (close(s1) < 0) {
      perror("ZZDUMMY_UN: CLOSE LISTENER SOCKET");
      exit(-1);
   }
   if ((rc = recv(s2, &size, sizeof(size), 0)) != sizeof(size)) {
      perror("ZZDUMMY_UN: INITIAL RECV");
```

```
      exit(-1);
   } else if (rc == 0) {
      printf("ZZDUMMY_UN: IR: RC = 0 => UNIX SOCKET IS DOWN\n");
      exit(-1);
   }
   if (send(s2, "Z", 1) <= 0) {
      perror("ZZDUMMY_UN: INITIAL SEND");
      exit(-1);
   }
   if (size > MAX_SIZE) {
      printf("ZZDUMMY_UN: SIZE = %d LARGER THAN MAX_SIZE = %d\n",
             size, MAX_SIZE);
      exit(-1);
   }
   bzero(buf, size);
   bytes_so_far = 0;
   for (;;) {
      if ((rc = recv(s2, buf, size, 0)) < 0) {
         perror("ZZDUMMY_UN: SUBSEQUENT RECV");
         exit(-1);
      } else if (rc == 0) {
         printf("ZZDUMMY_UN: SR: RC = 0 => UNIX SOCKET IS DOWN\n");
         exit(-1);
      }
      bytes_so_far += rc;
      if ((bytes_so_far / size) >= 1)
         if (send(s2, "Z", 1) <= 0) {
            perror("ZZDUMMY_IN: SUBSEQUENT SEND");
            exit(-1);
         }
      bytes_so_far %= size;
   }
}
```

```
/*****************************************************************************-
-                                                                            -
-    zzto_in.c                                                               -
-                                                                            -
-    Purpose - a small test program to time the delay in a "send" call       -
-                                                                            -
-    Call - zzto_in  remote_host [loops] [message_size]                      -
-                                                                            -
-    Defaults                                                                -
-        loops         100                                                   -
-        message_size  400                                                   -
-                                                                            -
-    Note - requires program "zzdummy_in" to serve as message target.  This  -
-           program supports INET sockets directly.  It does not require     -
-           support from an FTM system.                                      -
-                                                                            -
-****************************************************************************/
```

```
#include          <stdio.h>
#include          <sys/types.h>
#include          <sys/socket.h>
#include          <netinet/in.h>
#include          <netdb.h>
#include          <math.h>
#include          <sys/timeb.h>

#define            FTMPORT    9738
#define            MAX_SIZE   4096
#define            DEF_SIZE    400
#define            DEF_LOOPS   100


int             main(argc, argv)
   int             argc;
   char          **argv;

{
   char            msg[MAX_SIZE],
                   onebuf[1];

   int             i,
                   s1,
                   size = DEF_SIZE,
                   loops = DEF_LOOPS;

   struct sockaddr_in server;

   struct hostent *hp;

   struct timeb    time1,
                   time2;

   double          max = 0.0,
                   min = 2000000000.0,
                   delay,
                   avg,
                   sum = 0.0,
                   sumsqrs = 0.0,
                   stdev;


   if (argc < 2) {
      printf("ZZTO_IN: NO REMOTE HOST SPECIFIED\n");
      exit(-1);
   }
   if (argc > 2)
      sscanf(argv[2], "%d", &loops);
   if (loops < 1)
      loops = 1;
   printf("ZZTO_IN: NUMBER OF LOOPS = %d\n", loops);

   if (argc > 3)
      sscanf(argv[3], "%d", &size);
   if (size < 0)
      size = 0;
   printf("ZZTO_IN: MESSAGE SIZE = %d\n", size);

   if (size > MAX_SIZE) {
      printf("MESSAGE SIZE TOO BIG - LIMIT IS %d\n", MAX_SIZE);
      exit(-1);
```

```
    }
    if (size > 0) {
       msg[0] = 'Y';
       if (size > 2)
          msg[size - 2] = 'Y';
       for (i = 1; i < size - 2; ++i)
          msg[i] = 'X';
       msg[size - 1] = 0x00;
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
       perror("ZZTO_IN: GETHOSTBYNAME");
       exit(-1);
    } else if (hp->h_length != 4) {
       printf("ZZTO_IN: ILLEGAL NET ADDR LENGTH = %d \n",
              hp->h_length);
       exit(-1);
    } else if (hp->h_addrtype != AF_INET) {
       printf("ZZTO_IN: ILLEGAL ADDRTYPE = %d \n", hp->h_addrtype);
       exit(-1);
    }
    if ((s1 = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
       perror("ZZTO_IN: SOCKET");
       exit(-1);
    }
    bzero((char *) &server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = FTMPORT;
    bcopy(hp->h_addr, (char *) &server.sin_addr, 4);
    if (connect(s1, &server, sizeof(server)) < 0) {
       perror("ZZTO_IN: CONNECT");
       exit(-1);
    }
/*
-    make sure that zzdummy-in is active
*/
    if (send(s1, &size, sizeof(size), 0) != sizeof(size)) {
       perror("ZZTO_IN: INITIAL SEND");
       printf("ZZTO_IN: ZZDUMMY_IN MAY NOT BE ACTIVE\n");
       exit(-1);
    } else
       printf("ZZTO_IN: INITIAL SEND SUCCESSFUL\n");
    if (recv(s1, onebuf, 1, 0) != 1) {
       printf("ZZTO_IN: COULD NOT RECEIVE INITIAL ONE BYTE\n");
       exit(-1);
    }
/*
-    gather the statistics
*/
    ftime(&time1);
    for (i = 0; i < loops; ++i) {
       if (send(s1, msg, size, 0) != size) {
          perror("ZZTO_IN: SUBSEQUENT SEND");
          printf("ZZTO_IN: ZZDUMMY_IN MAY NOT BE ACTIVE\n");
          exit(-1);
       }
       if (recv(s1, onebuf, 1, 0) != 1) {
          printf("ZZTO_IN: COULD NOT RECEIVE SUBSEQUENT ONE BYTE\n");
          exit(-1);
       }
       ftime(&time2);
       delay = time2.time - time1.time
          + ((short) time2.millitm - (short) time1.millitm) / 1000.0;
```

```
      if (delay > max)
         max = delay;
      if (delay < min)
         min = delay;
      sum += delay;
      sumsqrs += delay * delay;
      time1.time = time2.time;
      time1.millitm = time2.millitm;
   }
   avg = sum / loops;
   if (loops > 1)
      stdev = sqrt((sumsqrs - (sum * sum / loops)) / (loops - 1));
   else
      stdev = 0.0;
   printf("MAX = %6.3f,   MIN = %6.3f,   AVG = %6.3f,   STDEV = %6.3f\n",
         max, min, avg, stdev);
   exit(0);
}
```

```
/*****************************************************************************
-                                                                            -
-    zzto_un.c                                                               -
-                                                                            -
-    Purpose - a small test program to time the delay in a "send" call       -
-                                                                            -
-    Call - zzto_un  [loops] [message_size]                                  -
-                                                                            -
-    Defaults                                                                -
-        loops         100                                                   -
-        message_size  400                                                   -
-                                                                            -
-    Note - requires program "zzdummy_un" to serve as message target.  This  -
-           program supports UNIX sockets directly.  It does not require     -
-           support from an FTM system.                                      -
-                                                                            -
-****************************************************************************/

#include          <stdio.h>
#include          <sys/types.h>
#include          <sys/socket.h>
#include          <sys/un.h>
#include          <math.h>
#include          <sys/timeb.h>

#define            FTMFAM    "/tmp/auvschi/zzmysocket.sck"
#define            MAX_SIZE    4096
#define            DEF_SIZE     400
#define            DEF_LOOPS    100


int             main(argc, argv)
   int             argc;
```

```
   char          **argv;

{
   char            msg[MAX_SIZE],
                   onebuf[1];

   int             i,
                   s1,
                   size = DEF_SIZE,
                   loops = DEF_LOOPS;

   struct sockaddr_un server;

   struct timeb    time1,
                   time2;

   double          max = 0.0,
                   min = 2000000000.0,
                   delay,
                   avg,
                   sum = 0.0,
                   sumsqrs = 0.0,
                   stdev;


   if (argc > 1)
      sscanf(argv[1], "%d", &loops);
   if (loops < 1)
      loops = 1;
   printf("ZZTO_UN: NUMBER OF LOOPS = %d\n", loops);

   if (argc > 2)
      sscanf(argv[2], "%d", &size);
   if (size < 0)
      size = 0;
   printf("ZZTO_UN: MESSAGE SIZE = %d\n", size);

   if (size > MAX_SIZE) {
      printf("MESSAGE SIZE TOO BIG - LIMIT IS %d\n", MAX_SIZE);
      exit(-1);
   }
   if (size > 0) {
      msg[0] = 'Y';
      if (size > 2)
         msg[size - 2] = 'Y';
      for (i = 1; i < size - 2; ++i)
         msg[i] = 'X';
      msg[size - 1] = 0x00;
   }
   if ((s1 = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
      perror("ZZTO_UN: SOCKET");
      exit(-1);
   }
   bzero((char *) &server, sizeof(server));
   server.sun_family = AF_UNIX;
   strcpy(server.sun_path, FTMFAM);
   if (connect(s1, &server, sizeof(server)) < 0) {
      perror("ZZTO_UN: CONNECT");
      exit(-1);
   }
/*
-    make sure that zzdummy_un is active
```

```
*/
   if (send(s1, &size, sizeof(size), 0) != sizeof(size)) {
      perror("ZZTO_UN: INITIAL SEND");
      printf("ZZTO_UN: ZZDUMMY_UN MAY NOT BE ACTIVE\n");
      exit(-1);
   } else
      printf("ZZTO_UN: INITIAL SEND SUCCESSFUL\n");
   if (recv(s1, onebuf, 1, 0) != 1) {
      printf("ZZTO_UN: COULD NOT RECEIVE INITIAL ONE BYTE\n");
      exit(-1);
   }
/*
-    gather the statistics
*/
   ftime(&time1);
   for (i = 0; i < loops; ++i) {
      if (send(s1, msg, size, 0) != size) {
         perror("ZZTO_UN: SUBSEQUENT SEND");
         printf("ZZTO_UN: ZZDUMMY_UN MAY NOT BE ACTIVE\n");
         exit(-1);
      }
      if (recv(s1, onebuf, 1, 0) != 1) {
         printf("ZZTO_UN: COULD NOT RECEIVE SUBSEQUENT ONE BYTE\n");
         exit(-1);
      }
      ftime(&time2);
      delay = time2.time - time1.time
         + ((short) time2.millitm - (short) time1.millitm) / 1000.0;
      if (delay > max)
         max = delay;
      if (delay < min)
         min = delay;
      sum += delay;
      sumsqrs += delay * delay;
      time1.time = time2.time;
      time1.millitm = time2.millitm;
   }
   avg = sum / loops;
   if (loops > 1)
      stdev = sqrt((sumsqrs - (sum * sum / loops)) / (loops - 1));
   else
      stdev = 0.0;
   printf("MAX = %6.3f,   MIN = %6.3f,   AVG = %6.3f,   STDEV = %6.3f\n",
         max, min, avg, stdev);
   exit(0);
}
```