Carnegie Mellon University
Software Engineering Institute

Assessment of CORBA and POSIX.21 Designs for FAA En Route Resectorization

19980514 117

B. Craig Meyers
Daniel R. Plakosh
Patrick R. H. Place
Mark Klein
Rick Kazman

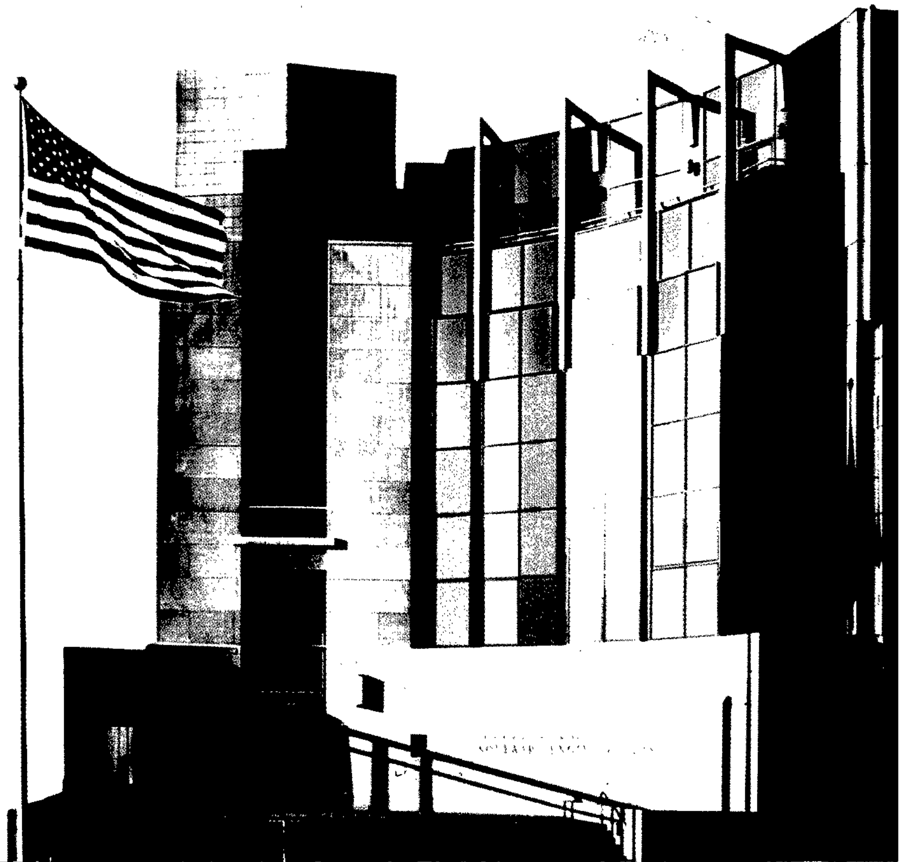
April 1998

DTIC QUALITY INSPECTED

SR

SPECIAL REPORT
CMU/SEI-98-SR-002

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of, "Don't ask, don't tell, don't pursue," excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

CMU/SEI-98-SR-002

Assessment of CORBA and POSIX.21 Designs for FAA En Route Resectorization

B. Craig Meyers
Daniel R. Plakosh
Patrick R. H. Place
Mark Klein
Rick Kazman

Dynamic Systems
Product Line Practices

April 1998

Carnegie Mellon University
Software Engineering Institute

Pittsburgh, PA
15213-3890

Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Jay Alonis, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright 1998 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Asset Source for Software Engineering Technology (ASSET): 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 or toll-free in the U.S. 1-800-547-8306 / FAX: (304) 284-9001 World Wide Web: http://www.asset.com / e-mail: sei@asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218 / Phone: (703) 767-8274 or toll-free in the U.S.: 1-800 225-3842.

Table of Contents

Executive Summary xv

1 **Introduction** 1

2 **Problem Statement** 3
2.1 Architectural Context 3
2.1.1 Overall Architectural Context 3
2.1.2 Scope of This Report 5
2.2 Intuitive Explanation of Sector Combination 6
2.3 Detailed Explanation of Problem 9
2.3.1 Fix Posting Areas 9
2.3.2 Sectors 12
2.3.3 Non-Geometric Considerations 13
2.3.3.1 Track Data 13
2.3.3.2 Flight-Plan Data 13
2.3.3.3 Data Blocks 13
2.3.3.4 Inbound List 13
2.3.3.5 Hold List 14
2.3.3.6 Conflict-Alert List 14
2.4 Component Combinations 14
2.4.1 Initial Sectorization for an En Route Center 14
2.4.2 Sector Combination 15
2.4.3 Assignment of a Sector to a Radar Display Console 15
2.4.4 Assignment of an FPA to a Sector 16
2.4.5 Assignment of One FPA to Another FPA 16
2.4.6 Non-Geometric Considerations 17
2.4.7 Atomicity of Operations 17

3 **Requirements Specification** 19
3.1 Scope 19
3.2 Assumptions 20
3.3 Software Requirements 21

3.3.1 System Management 22
3.3.1.1 General Requirements 23
3.3.1.2 State Data Management 23
3.3.1.3 Airspace Management 24
3.3.2 Radar Display Console Processing 30
3.3.2.1 Local Sector Airspace Management 30
3.3.2.2 Display of Track Data 32
3.3.2.3 Display Lists 32
3.3.3 Flight-Plan Management 33
3.3.3.1 Management of Flight-Plan State Data 33
3.3.3.2 Distribution of Flight-Plan Data 34
3.3.3.3 Flight-Plan Extrapolation 35
3.3.4 Track Management 36
3.3.4.1 Management of Track State Data 36
3.3.4.2 Distribution of Track Data 39
3.3.5 System Capacity Requirements 39
3.3.6 System Timing Requirements 40
3.4 Procedural Requirements 40

4 CORBA Approach 43
4.1 CORBA 43
4.1.1 Background 43
4.1.2 Architectural Overview 44
4.1.3 Real-Time Considerations 46
4.1.4 Communication Mechanisms 46
4.1.5 Additional Services 47
4.2 Presentation of Design Information 48
4.3 Basic Design Issues 50
4.3.1 Object Identification 51
4.3.2 Interface Determination 51
4.3.3 Design Transformation 52
4.4 Architectural Considerations 53
4.4.1 Chosen Architecture 53
4.4.2 Migration Considerations 55
4.5 Initial Design 56
4.5.1 System Management 58
4.5.2 Airspace Management 59

4.5.3 FPAs 62
4.5.4 Sectors 63
4.5.5 Consoles 65
4.5.6 Data Lists 67
4.5.6.1 Inbound List 67
4.5.6.2 Hold List 68
4.5.6.3 Conflict-Alert List 69
4.5.7 Track Management 70
4.5.8 Track Data 70
4.5.9 Flight-Plan Management 71
4.5.10 Flight Plans 72
4.5.11 Summary of Initial Design 73
4.5.11.1 Objects and Their Interaction 73
4.5.11.2 Sample Data Flows 75
4.6 Refinement of Initial Design 83
4.6.1 System Management 83
4.6.2 Airspace Management 84
4.6.3 FPAs 87
4.6.4 Sectors 87
4.6.5 Consoles 88
4.6.6 Data Lists 89
4.6.7 Track Management 90
4.6.8 Track Data 91
4.6.8.1 Object Considerations 91
4.6.8.2 Local or Distributed Management 92
4.6.8.3 Distribution of Track Data 93
4.6.9 Flight-Plan Management 98
4.6.10 Flight-Plan Data 100
4.6.11 Flight Plan/Track Correlation 100
4.6.12 Summary of Refined Design 101
4.6.12.1 Objects and Their Interaction 101
4.6.12.2 Mapping the Refined Design onto Hardware 103
4.6.12.3 Sample Data Flows 105
4.7 Implementation Concerns 111
4.8 Summary of CORBA Design 112

5 POSIX.21 Approach **113**
5.1 IEEE POSIX P1003.21 113
5.1.1 Background 113
5.1.2 Architectural Overview 114
5.1.3 Real-Time Considerations 114
5.1.4 Communication Mechanisms 115
5.1.5 Additional Services 116
5.2 Presentation of Design Information 118
5.3 Basic Design Issues 120
5.3.1 General Considerations 120
5.3.2 Use of Endpoints 121
5.4 Architectural Considerations 122
5.4.1 Chosen Architecture 122
5.4.2 Migration Considerations 123
5.5 Design 124
5.5.1 Design Components 124
5.5.1.1 System Management 124
5.5.1.2 Airspace Management 125
5.5.1.3 Track Management 126
5.5.1.4 Flight-Plan Management 129
5.5.1.5 Displays 131
5.5.2 Overall Design 132
5.5.2.1 Components and Their Interaction 132
5.5.2.2 Additional Comments 134
5.5.2.3 Sample Data-Flow Diagrams 134
5.5.3 Mapping the Design onto Hardware 142
5.6 Implementation Concerns 142
5.7 Summary of POSIX.21 Design 143

6 Assessment of Designs **145**
6.1 Overview of Principles 145
6.1.1 Architecture Trade-off Analysis 145
6.1.2 Why Use Architecture Trade-off Analysis? 147
6.1.3 The Architecture Trade-off Analysis Method 147
6.1.4 The Steps of the Method 148
6.2 Candidate List of Scenarios 152
6.3 Performance Assessment 152

6.3.1 Performance Considerations 152
6.3.2 A Performance Scenario 153
6.3.3 Performance Using CORBA 154
6.3.3.1 Scenario Realization 155
6.3.3.2 Performance Modeling 162
6.3.4 Performance Using POSIX 170
6.3.4.1 Scenario Realization 170
6.3.4.2 Performance Modeling 173
6.4 Modifiability Assessment 179
6.4.1 Brief Description of SAAM 179
6.4.2 Modifiability Using CORBA 180
6.4.2.1 Scenario Realization and Refinement 181
6.4.2.2 Scenario 1: Dynamic Sector Boundaries in CORBA 181
6.4.2.3 Scenario 2: Live Insertion in CORBA 182
6.4.2.4 Scenario 3: Have All Consoles Aware of All Data in CORBA 184
6.4.3 Modifiability Using POSIX 185
6.4.3.1 Scenario Realization and Refinement 186
6.4.3.2 Scenario 1: Dynamic Sector Boundaries in POSIX 187
6.4.3.3 Scenario 2: Live Insertion in POSIX 188
6.4.3.4 Scenario 3: Have All Consoles Aware of All Data in POSIX 189
6.4.4 Comparing CORBA to POSIX 189

7 Summary **191**

References **193**

Appendix A Acronyms **195**

Appendix B Glossary **197**

Appendix C Possible Sector Changes 205
C.1 Taxonomy of Operations 205
C.2 Operations on One Node 206
C.3 Operations on Multiple Nodes 207
C.4 Operations on Line Segments 208
C.5 Operations on a Geometric Entity 211
C.6 Summary 213

Appendix D Additional Requirements Specification 215
D.1 Adaptation Data Management 215

Appendix E Message Contents 221

Appendix F Specification of Loading Conditions 223

Appendix G Details of CORBA Approach 225
G.1 Description of Objects 225
G.2 Sample IDL Description 225

List of Figures

Figure 1	Overview of Proposed En Route Architecture	4
Figure 2	Sample Flight Path Through ARTCC	6
Figure 3	Adjacent Sector Example	7
Figure 4	Illustration of Sector Combination	8
Figure 5	Details of Sector Combination	8
Figure 6	Sample Fix Posting Area	11
Figure 7	An FPA with Exclusive Modules	11
Figure 8	Sector Combination	15
Figure 9	Assigning an FPA to a Sector	16
Figure 10	Assigning an FPA to Another FPA	16
Figure 11	En Route Evolution Path	19
Figure 12	Simplified Functional Data-Flow Model	22
Figure 13	Interaction of Client and an Object	44
Figure 14	CORBA Architectural Model	45
Figure 15	Notation for Use of CORBA Methods	49
Figure 16	Notation for CORBA Event Channels	50
Figure 17	Assumed CORBA Architecture	54
Figure 18	CORBA-Based Migration Architecture	55
Figure 19	Console-Object Relationships	57
Figure 20	Initial Object Specification for System Management	58
Figure 21	Initial Object Specification for Airspace Management	61
Figure 22	Initial Object Specification for a Fix Posting Area	63
Figure 23	Initial Object Specification for a Sector	65
Figure 24	Initial Object Design for Console Class	67
Figure 25	Initial Object Specification for an Inbound List	68

Figure 26	Initial Object Specification for a Hold List	69
Figure 27	Initial Object Specification for a Conflict-Alert list	69
Figure 28	Initial Object Specification for Track Management	70
Figure 29	Initial Object Specification for a Track	71
Figure 30	Initial Object Specification for Flight-Plan Management	72
Figure 31	Initial Object Design for Flight-Plan Object	73
Figure 32	Initial Design Objects and Interaction	74
Figure 33	Initial Design for Track-Update Data Flow	76
Figure 34	Initial Design for Flight-Plan Update Data Flow	78
Figure 35	Initial Data Flow for Sector Combination	80
Figure 36	Initial Design for Data Flow of Console-Object Failure	82
Figure 37	Refined Object Specification for System Management	84
Figure 38	Refined Object Specification for Airspace Management	86
Figure 39	Refined Object Design for Console Class	89
Figure 40	Refined Object Specification for Track Management	91
Figure 41	A Console Requesting Track Information	94
Figure 42	A Track-Data Management Object Transferring Track Data to a Console	94
Figure 43	Using Methods to Distribute Track Data	95
Figure 44	Use of CORBA Event Channels for Distribution of Track Data	96
Figure 45	Refined Object Specification for Flight-Plan Management	99
Figure 46	Object Specification for Correlation Processing	101
Figure 47	Refined CORBA Design	102

Figure 48	Centralized Hardware Mapping of CORBA Design	103
Figure 49	Distributed Mapping of CORBA Design	104
Figure 50	Refined Design for Track-Update Data Flow	105
Figure 51	Refined Design for Data Flow of Flight-Plan Update	107
Figure 52	Refined Design for Sector Combination	109
Figure 53	Refined Design For Console Failure	110
Figure 54	Architectural Context for POSIX.21	114
Figure 55	Sample POSIX.21 Design Notation	118
Figure 56	Additional POSIX.21 Notational Devices	119
Figure 57	Assumed POSIX.21 Architecture	123
Figure 58	POSIX Design for System Management	124
Figure 59	POSIX Design for Airspace Management	125
Figure 60	POSIX Design for Track Management	129
Figure 61	POSIX.21 Design for Flight-Plan Management	131
Figure 62	POSIX.21 Design for Display Console	132
Figure 63	Overall POSIX.21 Design	133
Figure 64	Track-Update Data Flow for POSIX.21 Design	135
Figure 65	Flight-Plan Data Flow Example	137
Figure 66	Sector-Combination Data Flow Diagram	139
Figure 67	Console-Failure Data Flow Example	140
Figure 68	Steps of the Architecture Trade-off Analysis Method	148
Figure 69	Conventions Used When Augmenting the Original Designs	154
Figure 70	Process/Thread Structure of System_Management	156
Figure 71	Process/Thread Structure of Airspace_Management	158

Figure 72 ORB and Airspace_Management 158
Figure 73 Process/Thread Structure of Console_Display_Object 161
Figure 74 Summary of Resectorization "String" of Computation 168
Figure 75 System_Management Threads When Using POSIX 171
Figure 76 Airspace_Management Threads When Using POSIX 172
Figure 77 Console Threads When Using POSIX 173
Figure 78 Summary of Resectorization "String" of Computation Using POSIX 177
Figure 79 CORBA Design with Realization of Scenario 1 182
Figure 80 CORBA Design with Realization of Scenarios 1 and 2 184
Figure 81 CORBA Design with Realization of Scenarios 1, 2 and 3 186
Figure 82 POSIX Design with Realization of Scenario 1 188
Figure 83 POSIX Design with Realization of Scenario 2 190
Figure 84 Moving an Existing FPA Node 206
Figure 85 Adding a Node to an FPA 207
Figure 86 Deleting a Node from an FPA 207
Figure 87 Coalescing Two Nodes Into One 208
Figure 88 Creating a New Line Segment 209
Figure 89 Moving a Line Segment 209
Figure 90 Deleting a Line Segment 210
Figure 91 Translation of a Line Segment 210
Figure 92 Translation and Rotation of a Line Segment 211
Figure 93 Scaling an FPA 211

Figure 94 Translating an FPA 212

Figure 95 Rotating an FPA 212

List of Tables

Table 1 System Capacity Requirements 39

Table 2 System Timing Requirements 40

Table 3 Estimated Number of Objects in Initial Design 75

Table 4 Comparing Methods and Event Channels for Distribution of Track Data 97

Table 5 Summary of CORBA Design-Object Characteristics 112

Table 6 Schedulability Model for Keyboard Latency Using CORBA 164

Table 7 Schedulability Model for Combine_Sectors in Airspace_Management 166

Table 8 Schedulability Model for Console_Display_Object 167

Table 9 Schedulability Model for Resectorization 169

Table 10 Schedulability Model for Keyboard "String" Using POSIX 174

Table 11 Schedulability Model for Combine_Sectors "String" 175

Table 12 Schedulability Model for Assign_Sector in Console Object 176

Table 13 Schedulability Model for Resectorization Using POSIX 177

Table 14 Schedulability Comparison for Resectorization 178

Table 15 Modifications Required to Satisfy Dynamic Sector Boundaries in CORBA 181

Table 16 Summary of Changes to Support Live Insertion in CORBA 183

Table 17 Modifications Required to Satisfy "Aware Consoles" in CORBA 185

Table 18 Modifications Required to Satisfy Dynamic Sector Boundaries in POSIX 187

Table 19 Summary of Changes to Support Live Insertion in POSIX 189

Table 20 Assumptions About Console Loading 224

Executive Summary

Background

Modernizing the En Route system presents major acquisition issues to the Federal Aviation Administration (FAA). At the present time, efforts are underway to upgrade the En Route system, primarily focusing on the host computer system. Some of the major issues include the following:

- What are the consequences of using different technologies and products?
- How can one assess the use of different technologies?

Resolving issues such as those above will affect the acquisition strategy for upgrading the En Route system. For example, the use of different technologies will influence the design character of the En Route system. This will consequently effect the ability to integrate different components and influence the amount of developed integration code that may be required. Clearly, the ability to optimize the acquisition strategy will have far-reaching consequences for the FAA and must be carefully considered.

This report addresses the use of different technologies and an architecture trade-off approach to a typical En Route system problem. We were requested to consider the problem of resectorization, i.e., the combination and decombination of sectors (and fix posting areas) during operation of an En Route center. Such capabilities may become desirable for an implementation of free flight. Two technologies have been applied to develop solutions to this problem, namely Common Object Request Broken Architecture (CORBA) and POSIX.21 (Portable Operating System Interface Standard). The former is based on an object-oriented model, while the latter is based on a message-passing model.

Approach

The context for this work is in a rearchitecture of the existing host computer system. The main functions of the current host are track management and flight plan management. The design presented here accounts for the distribution of functionality onto both hardware and software components. For both CORBA and POSIX.21, the designs were strongly influenced by both performance and fault-tolerant considerations.

Requirements for the resectorization problem were developed based on the existing host. To be clear about the problem considered, we developed a set of requirements (based on the PAMRI 1.3 [Peripheral Adapter Module Replacement Item] documentation). The problem was constrained to its essential features. For example, we did not consider the initialization of the En Route system or operations that can be performed in a training mode. However, the requirements do address distribution of track and flight plan data, as well as data associated with a sector, such as the inbound list and hold list.

Designs for both CORBA and POSIX.21 were developed and then refined. The initial CORBA design was based on a maximal object principle. This principle is useful for understanding the potential objects and the way in which those objects must interact. This allows the functional concerns to be separated from a choice of possible implementation strategies. The initial design contained over 2500 objects. The design was refined by using rough estimates of performance.

The initial POSIX.21 design was based on the identification of communicating objects (known as endpoints) and the way that they communicate (by the exchange of messages), and the mechanism for exchange of messages (such as unicast, multicast, or broadcast). This represents a different type of abstraction than that used in the CORBA design, yet there is still an emphasis on data abstraction, a prime characteristic of traditional object-oriented designs.

Each design was assessed by applying an architectural trade-off analysis (ATA) approach. The premise of ATA is that quality attributes can be assessed and trade-offs between them made by considering a system's architecture. The quality attributes considered in this report were modifiability and performance. For modifiability, the following scenarios were considered: (1) sector combination, (2) dynamic sector boundaries, and (3) console failure. The performance attribute was assessed by developing an analytic model for the time required to complete a system management operator request to combine two sectors. The model includes preemption, blocking, and execution effects.

Results

This report is not a comparison of CORBA and POSIX.21. There is the temptation to cast the results of this work as a comparison of CORBA and POSIX.21. To a limited degree this is true. However, only qualitative estimates are presented in the context of one problem. Such results do not lead to generalization. Both CORBA and POSIX.21 have strengths and weaknesses. It was not the purpose of this work to assess each design technology systematically in the context of the overall En Route system.[1]

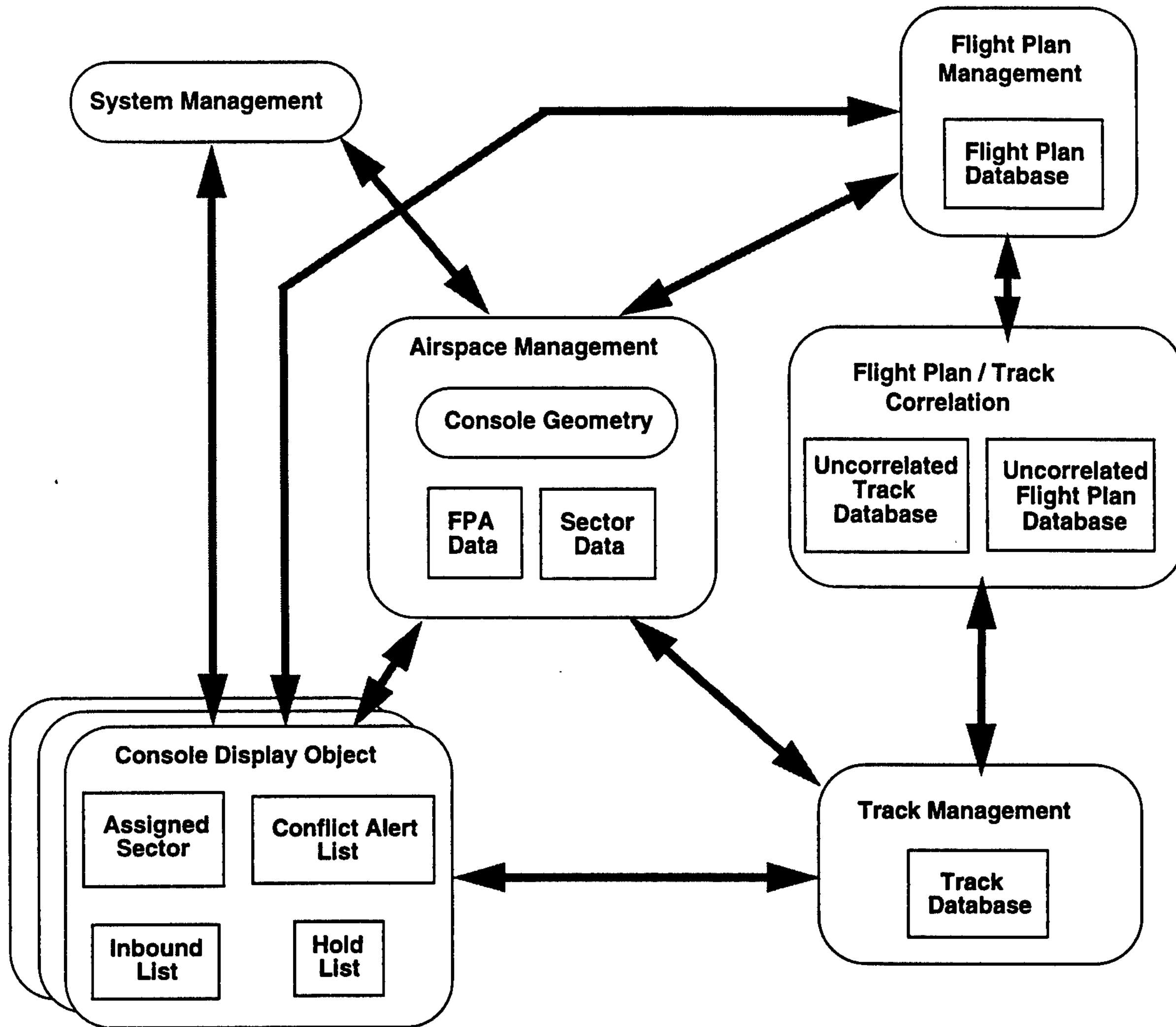
The development of requirements for the resectorization problem was complicated by potentially missing requirements. We found a number of problems with the current requirements specification. It quickly became apparent that (1) there are requirements that are implemented in a procedural manner, and/or (2) there are unstated requirements in the software documentation. An example of resectorization deals with adjacency of sectors in order for them to be combined. We would expect that there would be a software requirement that the sectors be adjacent in order for them to be combined, but we found no such requirement.

The designs exploited strengths of CORBA and POSIX.21. For example, the strengths of CORBA were support for object-oriented abstraction and support for heterogeneous systems. On the other hand, the strengths for POSIX.21 were flexible message-oriented design models and performance.

The CORBA design required significant refinement to meet performance and fault-tolerant requirements. One way to characterize the En Route system is as a collection of large numbers of potential objects, such as flights and tracks, with strong coupling between the objects. For example, conflict-alert processing is performed on the tracks to determine possible safety violations indicating the close coupling between track objects. The existence of close coupling implies communication between and among many objects. The current CORBA specification does not provide a one-to-many method invocation. Hence, the refinement of the initial design caused us to eliminate a track as a CORBA object, and tracks were subsequently encapsulated in a database. Decisions such as this caused the initial object-oriented design to lose much of its object character. A depiction of the final CORBA design appears below.

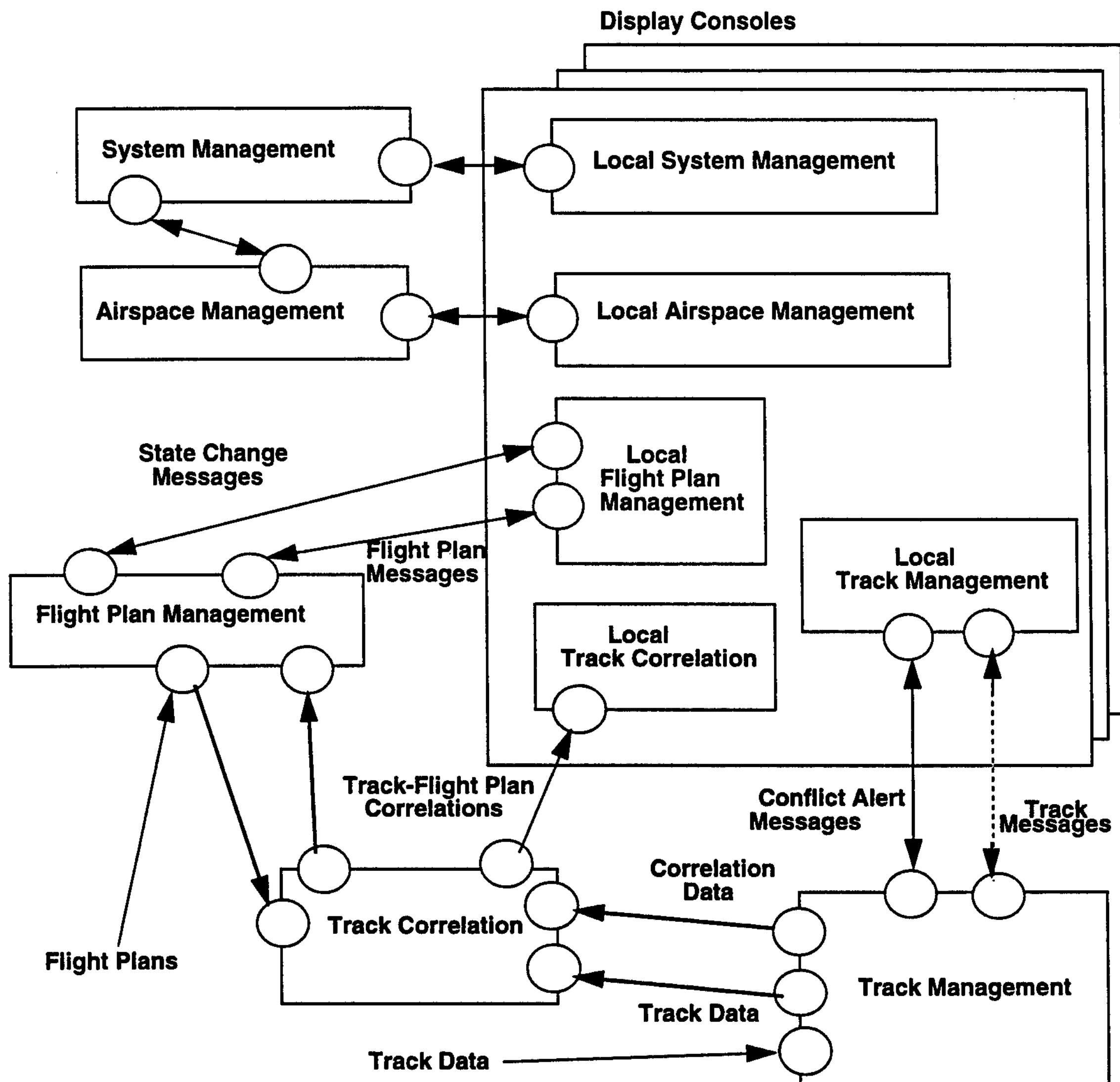
---

1. There is additional work that provides a comparison of CORBA and POSIX.21 with respect to support for real-time distributed systems communication. This is a joint effort between the SEI and MITRE and will be published as an SEI technical report, *A Comparison of CORBA and POSIX.21 With Respect to Real-time Distributed Systems Communica*tion (CMU/SEI-97-TR-15), B. Craig Meyers, Patrick R. H. Place, and Arkady Kanevsky. Pittsburgh, Pa.: Software Engineering Institute.



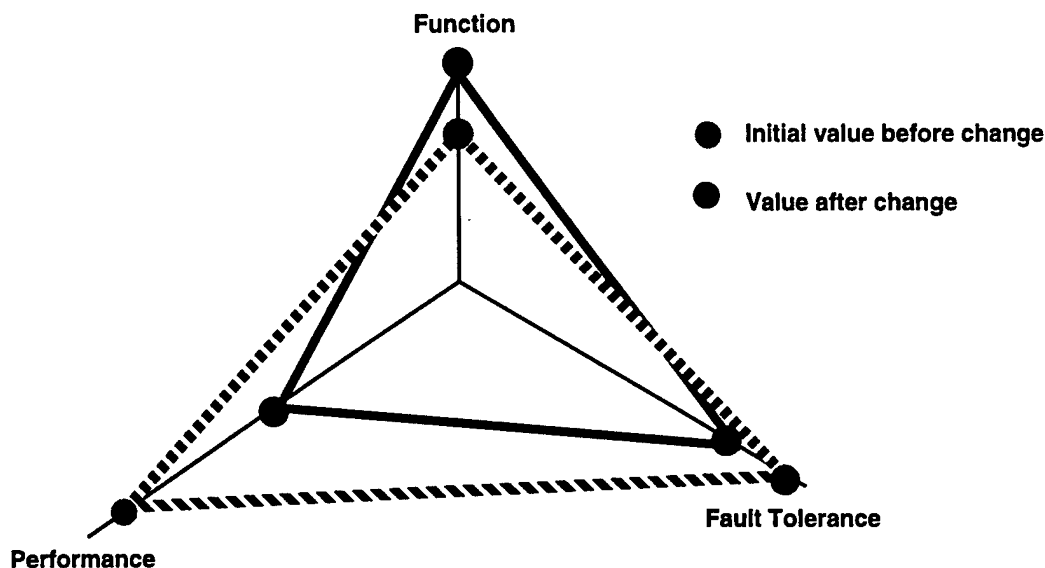
The POSIX.21 design exploited features that are useful for the real-time domain. For example, the POSIX.21 standard includes memory management (to eliminate multiple copies on data transfer), message priorities (integer and deadline), and a number of one-to-many communication mechanisms. The use of multiple communication models allowed for a greater distribution of state data, resulting in data (and processing) distribution onto replicated display

consoles. A consequence of this is that the time to restart a display console can be performed in an efficient manner. A depiction of the POSIX.21 design appears below.



The assessment procedure illustrated a number of technical points that showed the strengths and weaknesses of each technology considered. By assessing different scenarios, we were able to predict the consequences of system changes based on the particular scenario considered. For example, consideration of the sector recombination scenario showed that maintenance of global state data in a CORBA design would be more performance intensive than in a POSIX.21 design. The reason for this is the use of multicast in POSIX.21, thus permitting a more efficient communication mechanism than those currently available in CORBA.

Assessment of multiple attributes concurrently provided a deeper understanding of system issues. The principal attributes of concern were modifiability and performance. It is clear that a modification to the system will have performance consequences and that these attributes must be examined together to achieve a system view. For example, consider the ability to have each console aware of each track. The modification to both CORBA and POSIX.21 designs to accommodate this were almost identical. However, the two designs have dramatically different performance characteristics. In the POSIX.21 design, there was small impact on performance, but the CORBA design would be highly unlikely to meet performance requirements. The impact of considering multiple attributes is shown below. As illustrated, it may be possible to increase functionality, but decrease the performance and fault-tolerance characteristics of a design.



Recommendations

A formal specification of some selected En Route requirements should be performed. It is well known that the use of formal specification techniques will help in the specification of requirements. They provide for a clear and concise specification (in mathematical notation) and help to identify incomplete and inconsistent requirements. We believe a formal specification of some basic En Route functionality would help to improve understanding.[1]

1. The SEI is working with Dr. Valerie Harvey, a visiting scientist from Robert Morris College, who is developing a small formal specification for the resectorization problem.

In addition, our assessment of the current software requirements indicates that more modern practices could be applied to the specification of requirements. One example of this is increased use of state models and state-transition diagrams. Another example is to focus on a data-centric approach. For example, what are the states and attributes of a sector? It would be interesting to develop a prototype document that could be hosted on the Web.[1]

The case of a hybrid design approach, incorporating both CORBA and POSIX.21 components, should be considered. This report focuses on two extreme design approaches, one entirely based on CORBA and one entirely based on POSIX.21. It may be the case that the most appropriate design is one that uses *both* CORBA and POSIX.21. We term this a *heterogeneous* design approach. A fundamental question in this context is the amount of coupling between a CORBA and POSIX.21 design. That is, would an overall design really be two loosely-coupled designs (one CORBA and one POSIX.21)? If these approaches must interact, how may this best be accomplished? For example, what would it take for a CORBA and POSIX.21 implementation to interoperate? A myriad of interesting technical issues are associated with this issue.

The architecture tradeoff analysis should be extended, including development of a *Guide to Architectural Performance Analysis (APA).* The architectural analysis approach was useful in assessing the designs in this report. Further work is warranted in this area to assess the interaction of multiple attributes. For example, if a modifiability scenario is proposed, what are the performance consequences of the considered design changes?

The architectural performance analysis applied in this report was especially useful as a means to assess competing designs for the same problem. We believe that this approach should be presented as a guide, discussing the purpose and a method for conducting different analyses. The existence of such a document could have several benefits to the FAA. For example, it can be recommended (or required) that a contractor apply this approach when responding to a request for proposal (RFP). It therefore becomes an evaluation criteria for an RFP submission. Naturally, it would also be necessary, as part of this task, to develop suggested RFP language for the use of the APA approach. Another potential advantage would be that we recommend an FAA contractor be tasked to perform an APA in the context of the host upgrade. This would help in understanding the potential changes (and cost) that may be necessary in upgrading DSR.

---

1. For example, we have considered the notion of an annotated software requirements specification (SRS) that would permit a reader to easily move through the document and obtain background information about a particular requirement (such as rationale). Hosting such a document on the Web would be an interesting approach to distributing the documents.

The performance assessment should be extended to include quantitative results. This report does not include quantitative results for a number of reasons. First, quantitative results are both implementation and platform dependent. Second, a quantified model will most likely require values for commercial off-the-shelf (COTS) products, which may not be available. Continuing work in this direction may require only ball-park estimates, and coupled with sensitivity analysis, may shed further insight into the design.

A goal of performance analysis is the ability to predict the execution characteristics for a particular design quantitatively. To achieve this requires performance characteristics of the system components, including COTS components. There are interesting challenges to developing a performance model that includes COTS components. For example, a number of the performance characteristics may not be provided by a vendor and thus need to be determined. This may naturally lead one to consider the development of benchmarks for CORBA and POSIX.21. The existence of such benchmarks would aid in the selection of implementations that meet necessary performance characteristics for inclusion in the En Route architecture.

Other scenarios (e.g., hand-off, flight-plan management) should be considered. The more scenarios that are assessed, the more confidence one has about the design. It would be interesting to see the additional changes to the design as a result of including more scenarios. Both hand-off and flight-plan management are directly related to the resectorization problem.

Support for development of real-time CORBA and POSIX.21 should be continued. FAA and other organizations have contributed to the development of CORBA and POSIX.21. There are several reasons to continue this support. One is to keep abreast of the development of the specifications (and products). Another, more important, reason is to try to influence the standard so that it is appropriate for the FAA domain. Finally, there is the intriguing possibility that a real-time CORBA could be developed on top of a POSIX.21 implementation.

1 Introduction

The acquisition and evolution of major software-intensive systems requires that a number of complex and challenging questions be considered. One of these questions deals with the technology that will be used in the system. The choice of a technology can have implications in a number of areas, such as

- performance and/or fault tolerance of implementations system components
- the degree to which the system can be modified as part of its evolution
- the degree to which different technologies may be used to integrate other system components

The Federal Aviation Administration (FAA) is in the process of major program acquisitions to modernize both hardware and software for the En Route centers. As part of this acquisition effort, it must address issues such as those posed above. The purpose of this report is to address the following issues:

- Illustrate the application of CORBA and POSIX.21 as two different technology choices for the FAA En Route system. The former supports an object-based approach, while the latter is a message-based model.
- Demonstrate the application of a systematic approach to comparing different design approaches to the same problem.

In comparing different technologies, there are a number of approaches that one may take. Some examples include assessments based on

- specifications
- implementations
- model problems based on both specifications and implementations

Specification-based assessment examines a specification with respect to some set of criteria. An example of this approach, for Common Object Request Broker Architecture (CORBA) and POSIX.21 (Portable Operating System Interface Standard), is presented in *A Comparison of CORBA and POSIX.21 From a Real-Time Communication Perspective* [Meyers 97]. Assessment of an implementation examines one (or more) implementations of a technology specification. The assessment of implementations is similar to the application of different types of benchmarks. An assessment can also be based on model problems, the advantage being that it is performed in the context of an application. In this report, we will use a model-problem

approach based on a specification of an architecture to illustrate how two different technologies could be applied to the same problem. Performing an assessment of candidate architectures *early* in the development phase has certain obvious advantages.

The model problem that will be considered in this report is that of *sector combination*. This allows for changing the airspace for which an air traffic controller is responsible during system operation. The existence of such a capability helps to balance the load on controllers. However, in the context of a future En Route system, combining sectors or performing other operations, such as dynamically changing the boundary of a sector, may assist in the implementation of a free-flight capability.

This report is organized in the following manner; Section 2 provides a discussion of the problem. In Section 3, we provide a discussion of the requirements for the resectorization problem. The design approaches for CORBA and POSIX.21 are then presented in Sections 4 and 5, respectively. The assessment of these different approaches is discussed in Section 6. A brief summary of the work appears in Section 7. A number of appendices accompany this report, including a glossary, list of acronyms, and information relevant to the details presented in the text of the report.

We would like to thank Bill Wood and Valerie Harvey for discussions related to this work.

2 Problem Statement

2.1 Architectural Context

2.1.1 Overall Architectural Context

It is helpful to place the present work in the context of an overall architectural structure. To illustrate this, we have chosen the architectural view developed by a special study, commissioned by the FAA. The proposed architecture [FAA 95a] is presented in Figure 1 below.

The boxes in Figure 1 denote the major subsystems that comprise an En Route Center. The rings in the figure denote a mechanism to achieve connectivity among the indicated subsystems. It does not necessarily denote a physical ring; other hardware representations may achieve the same goals. Figure 1 includes two networks interconnected by a gateway. The reason for two networks is for fault-tolerant considerations.

The functionality of the various subsystems shown in Figure 1 is described below:

- *Air Traffic Management:* provides predictions of potential separation issues; helps controllers to optimize air-traffic flow; helps controllers plan for resolution of potential problems
- *Communication Subsystem:* provides for data transfer (both send and receive) to other airspace facilities; provides assistance to controllers for aircraft control and clearances
- *D-side Controller:* manages controller data display
- *Display Management:* presents integrated view and allows for monitoring and planning for air traffic controllers. (This function also serves as a directory-like agent to map sectors onto physical addresses for display consoles.)
- *Flight-Data Processing:* receives and processes flight plan information; provides capability to predict aircraft positions
- *System Management (M&C; Maintenance and Control):* provides for overall system management
- *Prototype Gateway:* provides a mechanism to incorporate prototypes in an operational system
- *R-side Controller:* manages controller radar display
- *Surveillance Data Processing:* receives and processes real-time data on air tracks; develops predictions based on real-time data and identifies potential problems to controllers

- *Weather:* receives weather information (observations and forecasts) and provides this information to other subsystems

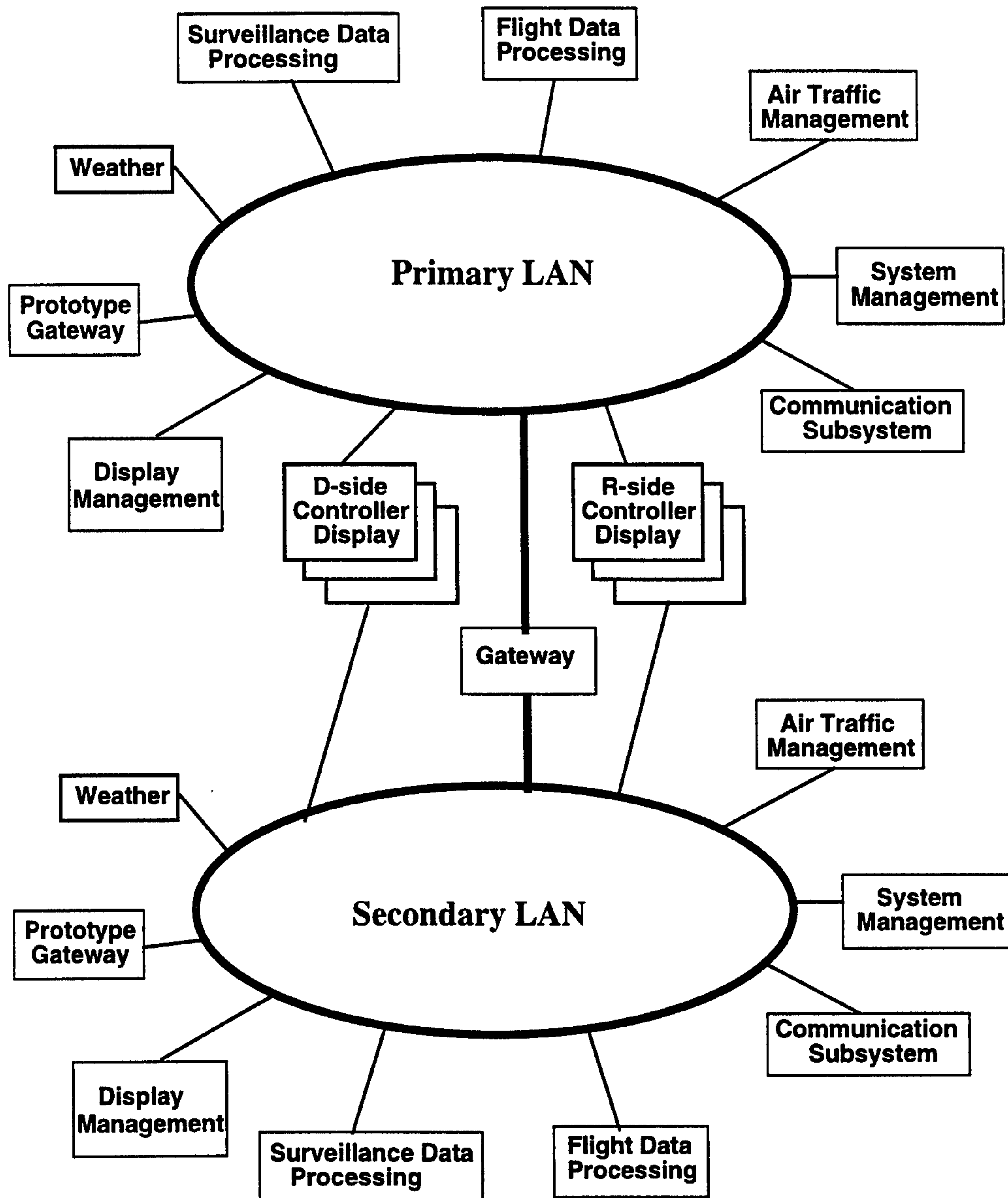


Figure 1: Overview of Proposed En Route Architecture

The preceding is a simplified overview of the various subsystems *proposed* for the En Route architecture.

The currently deployed system includes a host computer system (HCS) which interfaces with various display devices. The principal functions of the host are to

- manage track data
- manage flight-plan data
- manage communications with external systems

Clearly, the host is the central computer system of the current architecture.

2.1.2 Scope of This Report

For the purpose of this report, it will not be necessary to include all of the elements indicated in the preceding figure. The scope of this report is restricted by the following:

- We assume that the current HCS, currently implemented in a single processor, will change to accommodate modernization and increased functional changes.
- We assume that there may need to be changes to the current display system replacement (DSR) consoles with which a revised host would interface.
- The degree of change, for either the host or DSR, with respect to the use of either CORBA or POSIX, is a function of the need for change and is discussed later in this report.

One of the basic issues that must be faced by the FAA is how the host will change in the future. For example, will it remain a centralized architecture, or will it move to a distributed architecture? Influencing the architectural decision is the degree to which commercial off-the-shelf (COTS)-based components could be applied to the overall En Route architecture. Further discussion of the evolutionary aspects of the overall En Route system are discussed in Section 3.1.

This report will address the problem of *sector combination* in an En Route Center. This problem was posed by FAA management as a typical problem that may be relevant for the future En Route Center.[1] This report was motivated by comparing how a CORBA approach would

1. An earlier description of this problem was addressed in a limited context. Meyers, B. Craig & Place, Patrick R. H. *The Use of IEEE Draft Standard 1003.21 Real-Time Distributed Systems Communication in an FAA En Route Architecture* (CMU/SEI-Special Report). Pittsburgh, Pa.: Software Engineering Institute, March 1, 1997.

compare with the POSIX.21 approach. The use of sector combination may then be viewed as a model problem, to be examined using two different technologies.

2.2 Intuitive Explanation of Sector Combination

In this section we present an intuitive description of the problems related to sector combination. The flight of an aircraft is controlled by a number of facilities. One such facility, called an En Route Center (denoted ARTCC for Air Route Traffic Control Center), is responsible for controlling the flight as it crosses airspace between airports. Thus, an ARTCC may be viewed as a collection of airspaces, known as sectors. Associated with a sector are air traffic controllers who maintain the following positions:

- R-position: a controller who manages radar data
- D-position: a controller who manages data for the R-position controller

An example of the airspace managed by an ARTCC is presented in Figure 2 below.

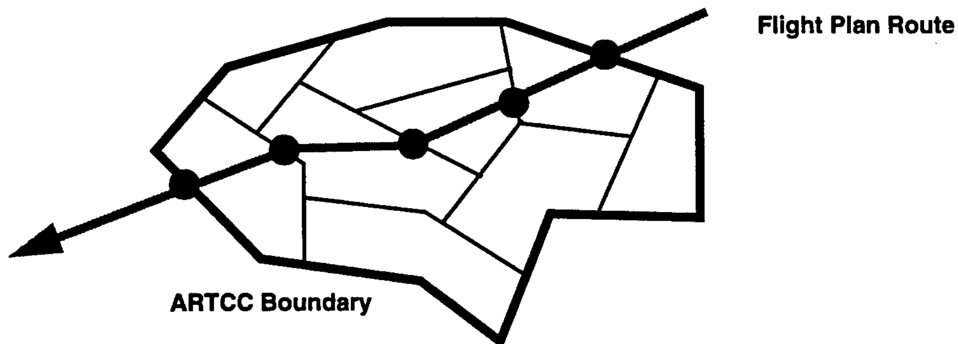


Figure 2: Sample Flight Path Through ARTCC

This figure illustrates an expected path of an aircraft through an ARTCC airspace, as denoted by the shaded arrow. The filled circles denote *handoff* points, either from one ARTCC to another ARTCC, or from one sector to another sector within the same ARTCC. Note that the anticipated path crosses a number of sectors that comprise the ARTCC topology. Each display console maintains an inbound list that indicates the expected time at which a controller must be made aware of a flight that will enter that controller's sector from another sector. The distribution of flight-plan information related to the inbound list is sequential in nature. That is, first the sector that will receive control of the flight is notified. Then, as the flight comes close to the next sector along the expected flight path, that sector is notified. This process continues until

the flight transits from one ARTCC to an adjacent ARTCC. Each such transition (either ARTCC to ARTCC, or sector to sector) involves a handoff of the flight.

As part of handing off an aircraft as it traverses the airspace managed by an ARTCC, it is necessary to determine the sectors that are adjacent to a given sector. An illustration of adjacent sectors is shown in Figure 3 below.

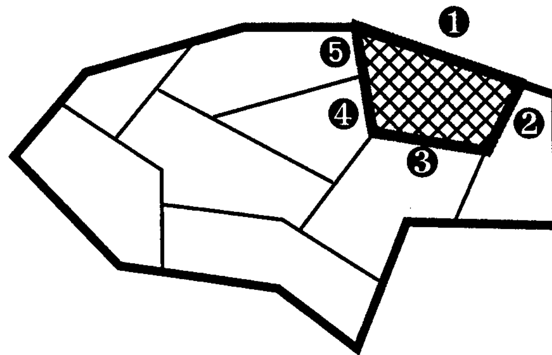


Figure 3: Adjacent Sector Example

In Figure 3, the sector that appears crosshatched has five adjacent sectors. The boundary indicated by ❶ is adjacent to an external facility. Note that one of the edges of the crosshatched sectors is adjacent to two sectors (namely those denoted ❹ and ❺). Knowledge of adjacent sectors and/or other En Route Centers is especially relevant if sector boundaries are dynamic (for example, if a controller could change the boundary of a sector).

The geometry (and other aspects) of a sector are specified as *adaptation data*. These data uniquely define a given ARTCC. Since ARTCCs differ in sector geometry, and many other factors, the use of adaptation data provides a means to configure a particular facility through data that are loaded as part of system initialization.

In the case of a sector combination, two (or more) existing sectors are combined to form a new sector. This is illustrated in Figure 4.

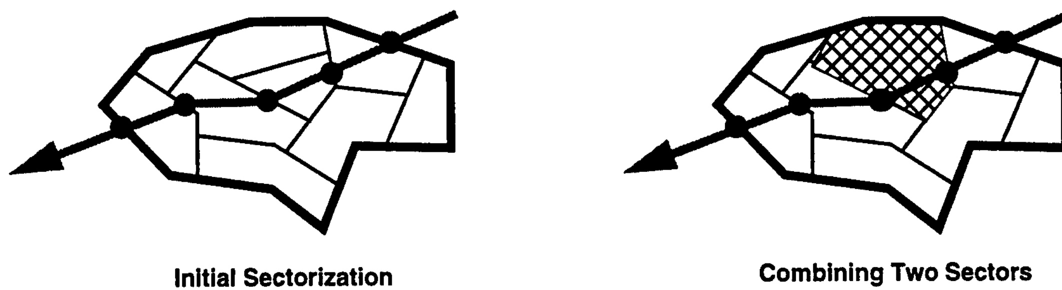


Figure 4: Illustration of Sector Combination

The result of the combination is to produce a new sector, as indicated by the shading on the right-hand side of Figure 4.

There are different ways in which the combination can be accomplished. We introduce the concept of an active and passive sector according to the following definitions:

- *active sector:* a sector in which a controller is currently providing control of aircraft
- *inactive sector:* a sector in which a controller is not currently providing control of aircraft.

Thus, it is the controller at the active sector who manages the new sector. This arrangement is shown in Figure 5.

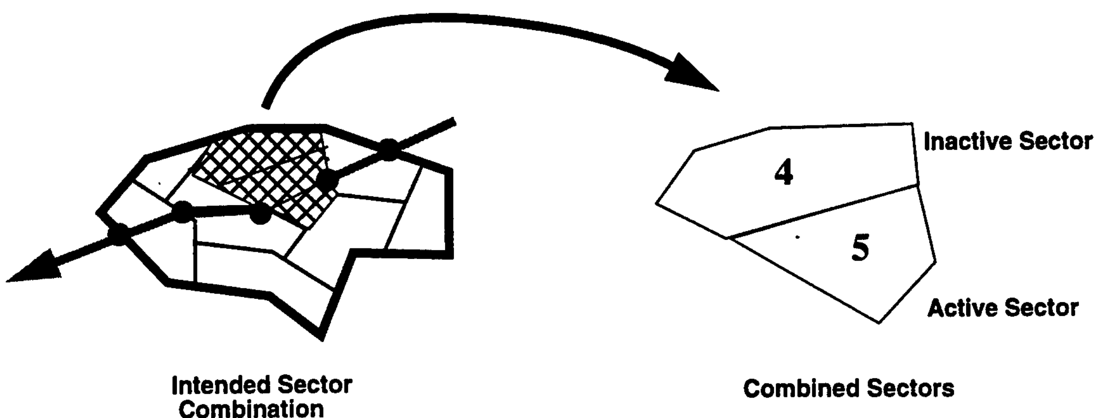


Figure 5: Details of Sector Combination

The sectors that are crosshatched in Figure 5 are intended to be combined into one sector. The resulting combination, involving sectors 4 and 5, is shown at the right of Figure 5. Sector 5 is designated as the active sector, while sector 4 is designated as an inactive sector.

To illustrate how sector combination could take place, we consider the following procedural semantics. Doing so helps illustrate the roles of the active and passive sectors:

1. Assume there exists a set of sectors, each of which is active. This means they are all managing flights within their own airspace.
2. A choice is made to combine two sectors. The choice is procedural in nature and initiated by an operator. One of the initial sectors is chosen to be the active sector for the resulting combination; any others will become inactive sectors.
3. A message is sent to the affected consoles indicating that the sectors should be combined. The transfer of this message is initiated by a system management function.
4. Upon receipt of the message, the operational consoles that received the message initiate processing to transition to the indicated state. This means that the active sector will assume control of aircraft in the sector(s) designated to be inactive. Correspondingly, the sector(s) designated to be inactive will no longer perform control of aircraft in their airspace.

The conditions for which a sector combination is permitted could involve many considerations. For example, if a sector is designated to become inactive and that sector is in the process of handing off a flight, we would expect that the handoff would be completed before the sector is transitioned to the inactive state.

2.3 Detailed Explanation of Problem

The preceding discussion was a simplified introduction to the problem considered in this report. We now consider a more detailed presentation of the problem.

2.3.1 Fix Posting Areas

The fundamental unit of airspace that is managed by an ARTCC is a *fix posting area* (FPA). The following types of FPAs are defined:

- *En Route airspace:* An FPA that defines a unit of airspace in an ARTCC that is not used for the functions specified below. This represents the majority of cases for an FPA definition.
- *Approach control FPA:* An FPA that contains an approach-control facility, commonly referred to as a tower facility.

- *ARTS Approach control:* An FPA that contains an ARTS approach-control facility, commonly referred to as a TRACON.
- *No_Airspace:* An FPA that has no airspace associated with it, defined without an altitude. This type of FPA is used to describe postings for flow control and air management information service.

An FPA may also have attributes that indicate characteristics of the FPA. These include[1]

- a focal point fix (FPF), which is required for each domestic FPA (except those of type no_airspace), used for posting of direct route flights
- An entry point posting indicator (EPPI), which is used for direct route flights
- A direct route priority indicator (DRPI), which is used for direct routes
- The major and minor airways that are contained within an FPA
- an indication that the FPA is considered for oceanic processing
- an indication of a wind station, used to report wind velocities
- an indication if there is no radar coverage for the FPA
- information about adjacent FPAs
- information about map(s) associated with the FPA

An FPA is a three-dimensional volume of airspace that is defined in terms of horizontal[2] line segments, each of which has an altitude. The line segments that comprise an FPA form a polygon when viewed in the horizontal and vertical planes.The polygon may be either convex or concave. The points that define the line segments are called *nodes*, which are described in terms of latitude and longitude. Figure 6 illustrates a simple FPA.

It is also possible for an FPA to contain other FPAs. This is done to define exclusion regions. In this case, the structure of an FPA becomes more complex. For example, Figure 7 illustrates the case where one FPA contains other FPAs [FAA 95g].

1. This information is taken from pages 2-9 through 2-25 of FAA specifications [FAA 95g].
2. The term *horizontal* means parallel to the surface of a planar earth. This is the definition for the context of a non-Oceanic En Route Center.

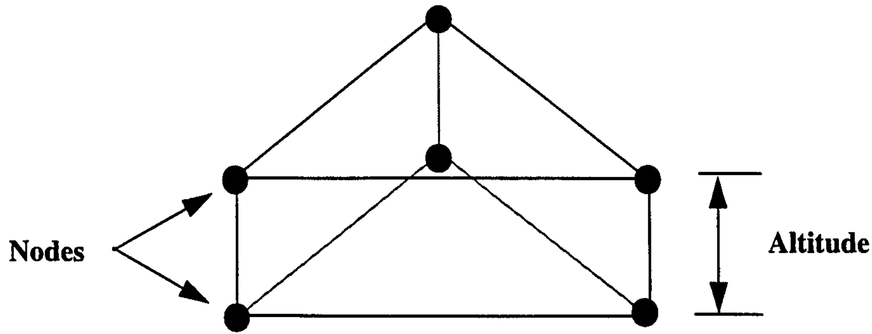


Figure 6: Sample Fix Posting Area

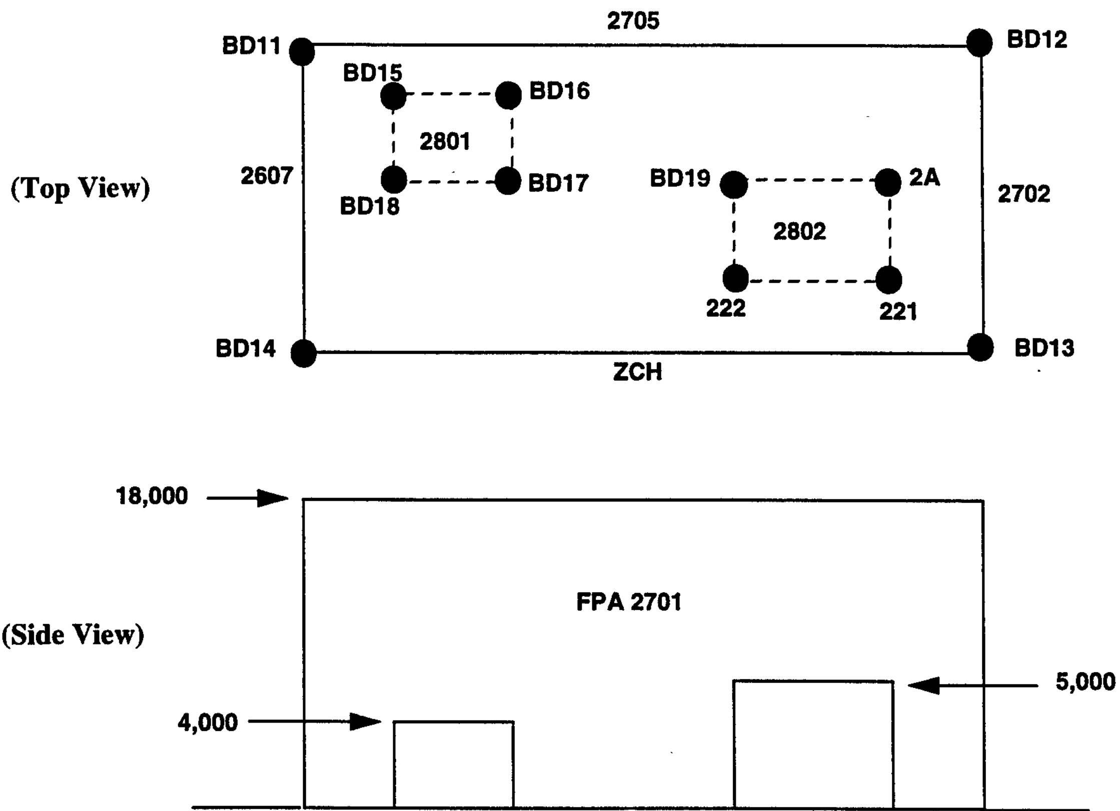


Figure 7: An FPA with Exclusive Modules

FPAs are defined as part of adaptation data. The basic information used to describe the FPA includes the following:

- a unique identifier
- line segments, defined in terms of nodes

- the altitude of each line segment
- each adjacent FPA(s) and/or adjacent ARTCC

In addition to the above data, the attributes of an FPA (discussed above) are also specified as part of adaptation data.

2.3.2 Sectors

A *sector* is a geographical area, having specified altitude limits, that is capable of being displayed on one (or more) display consoles. A *sector airspace* is a set of one or more contiguous FPAs that constitute a specified sector. A sector is identified by either a two-digit integer or an identifier.

Sectors are defined as part of adaptation data.[1] There are two major elements to the definition of sector adaptation data. The first is information concerning the configuration of a sector by assigning a set of devices to the sector. For example, the types of devices that may be assigned to the sector include

- flight strip printers
- D-position consoles (keyboard and display)
- R-position consoles (keyboard and display)

The second part of sector adaptation data is defined in terms of a *sector plan record*.[2] A set of sector plans is defined, one of which is called the *basic sector plan*. The basic sector plan is that plan in which each FPA is attached to the sector with the same number as the first two digits of the FPA identifier. The other sector plans are defined as modifications of the basic sector plan. For example, in sector plan 7, sector 12 could be defined as the set of FPAs associated with sector 12 in the basic sector plan; in addition, a number of other FPAs could be assigned to sector 12. The assignment of the FPAs to one sector also implies that the FPAs are removed from the default associated sector. In addition, one sector could be assigned to another sector as part of a sector plan.

---

1. See page 14-1 of FAA specifications [FAA 95d].
2. See pages 5-24 through 5-26 of FAA specifications [FAA 95g].

2.3.3 Non-Geometric Considerations

The previous discussion was oriented toward the geometric aspects of a sector and an FPA. From the perspective of an air traffic controller, there are the data associated with a sector. These data include

- track data
- flight-plan data
- data blocks
- inbound list
- hold list
- conflict-alert list

2.3.3.1 Track Data

A track refers to a computer-generated representation of an aircraft position and velocity. It may also contain other state data (such as the type of aircraft) and the temporal history of the location of the track. Track data are displayed and continuously updated on a display console. A track is sometimes denoted by a computer ID (CID).

2.3.3.2 Flight-Plan Data

A flight plan is a set of information about a flight. Flight plans are defined for an aircraft before it departs and can be modified during flight. Information associated with a flight plan includes, for example, a flight plan identification, assigned altitude, and route of flight.

2.3.3.3 Data Blocks

A data block refers to the display of information on a controller screen for a specified track. The information can include a symbol denoting the track position, indication of the track velocity, and numeric data about the state of the track (such as the altitude and velocity).

2.3.3.4 Inbound List

An inbound list contains a list of aircraft that are expected to enter a sector from either an adjacent facility or a tower facility that is contained within a particular ARTCC. An aircraft is added to the list for a particular sector when it is within a specified time interval of crossing the receiving sector's boundary.

2.3.3.5 Hold List

A hold list contains information about those flights that are placed in a hold state. The hold is defined with respect to a *hold fix* (a point in space). A flight can be added or removed from this list as a result of an operator action or receipt of a message from another facility.

2.3.3.6 Conflict-Alert List

A pair of tracks are said to be in conflict if there is an indication of an immediate, current, or impending violation of adapted spatial separation of aircraft. Such separation can be in a lateral direction or vertical direction between aircraft. If two tracks are detected to be in conflict, they are placed on the conflict-alert list for one, or possibly more, display consoles.

2.4 Component Combinations

We use the generic term component here to describe either a sector or a fix posting area. There are several combinations of components that are possible. Each of these is discussed in the following subsections. These combinations are accomplished through the use of the *Restore Message* (CS message).[1]

2.4.1 Initial Sectorization for an En Route Center

Before discussing possible combinations of sectors and/or FPAs it is important to understand how the initial state of an En Route Center is specified. This is achieved through the use of a *sector plan.* A sector plan is used to specify an initial configuration of a facility via adaptation data. One such plan is known as the basic sector plan. During initialization, a sector plan is defined for a facility. It may be either the basic sector plan or some other sector plan which we shall denote as a derived sector plan. A derived sector plan is a specification of only the modifications of the basic sector plan.

During the course of operations, it is possible to change the sectorization of a facility by applying a sector plan, known as *resectorization*. When this happens, the appropriate changes to the current sectorization are performed. This would result in controllers controlling different airspace.

---

1. See pages 6-37 through 6-42 of FAA specifications [FAA 95b].

2.4.2 Sector Combination

The simplest combination that is currently permitted is that where two (or possibly more) sectors are combined.This case is shown in Figure 8 below and is analogous to the discussion in Section 2.2. Initially, there were two sectors, each based on certain FPAs. The FPAs are denoted by the dashed lines in the figure.

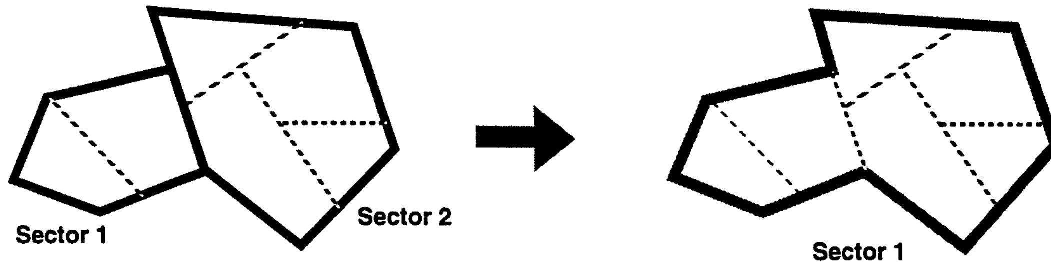


Figure 8: Sector Combination

The result of the combination is that sector 1 will become the active sector, while sector 2 becomes an inactive sector. This implies that the air traffic controller who was initially controlling sector 1, will now also be controlling the airspace defined by sector 2.

2.4.3 Assignment of a Sector to a Radar Display Console

Another type of operation, which is not explicitly a combination of either a sector and/or FPA, is the ability to assign a radar display console to a specified sector. When this happens, the display of the specified console is changed to display the specified sector. As a result of this change, the controller now assumes control of the aircraft in the specified sector.

2.4.4 Assignment of an FPA to a Sector

Another possible combination is that where an FPA is assigned to a sector. This case is illustrated in Figure 9 below. The figure shows the result of combining FPA 0201 with sector 1.

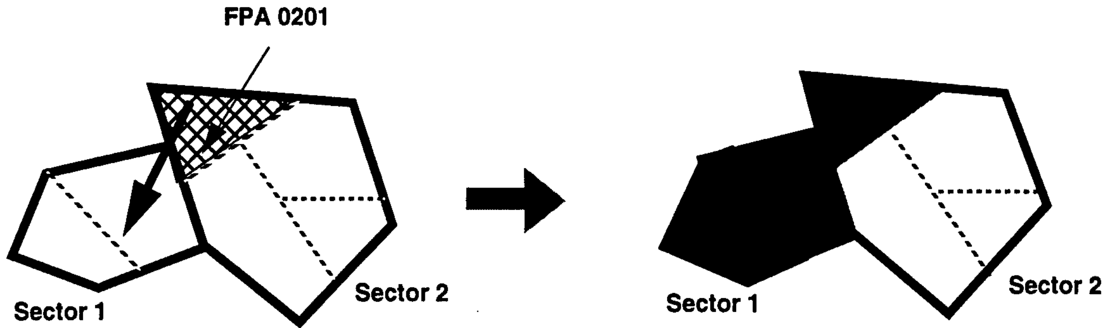


Figure 9: Assigning an FPA to a Sector

The result of the assignment of the FPA to the sector is shown on the right-hand side of Figure 9. The shaded area represents the new airspace which constitutes sector 1. This means that the airspace denoted by the shaded region will now be controlled by the controller in sector 1. The remaining airspace will continue to be under the control of sector 2.

2.4.5 Assignment of One FPA to Another FPA

The last combination that is currently permitted is the case where an FPA is assigned to another FPA. This case is shown in Figure 10 below. The sector that is crosshatched on the left side of the figure is intended to be combined with the indicated FPA.

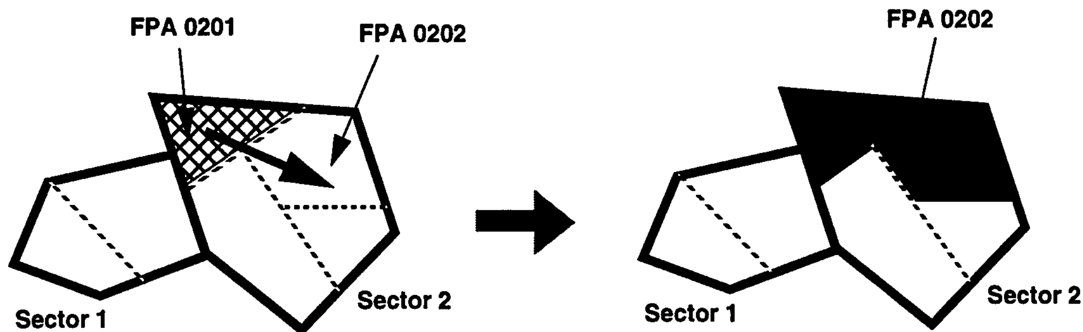


Figure 10: Assigning an FPA to Another FPA

Initially, FPAs 0201 and 0202 are part of sector 2. The figure illustrates the case where FPA 0201 is assigned to FPA 0202. The result of the assignment is shown on the right side of Figure 10. The shaded area denotes FPA 0202, the result of the assignment. After the assignment, FPA 0202 will be the *primary* FPA, and FPA 0201 will be the *subjugate* FPA.

Note that the assignment of one (or more) FPA to another FPA could happen in a number of contexts. These include

- the case where all FPAs are in the same sector, as indicated above
- the case where one FPA is in one sector and another FPA is in another sector

2.4.6 Non-Geometric Considerations

The preceding discussion has focused only on the geometric aspects of an FPA or sector. A number of data elements that are relevant to the control of aircraft were noted in Section 2.3.3. It is important to understand that when a sector and/or FPA change, there are associated changes with the data elements associated with the sector.

For example, each sector has an associated hold list. If two sectors are combined, it is necessary to combine the hold lists of each individual sector. A similar action must be performed for the inbound lists and conflict alert list. Also, flight plan information is distributed to certain display consoles. Hence, when two sectors are combined, there may be a change in the distribution of flight plan data.

2.4.7 Atomicity of Operations

Consider the case where two sectors are being combined. A basic question is whether the combination should be performed in an *atomic* manner. That is, either the sector combination will complete, or no combination will take place. An alternative is to allow the sector combination to proceed in parallel, but if there is a failure, the desired combination may not be realized. If a dynamic change to an FPA or sector should be performed in an *atomic* manner, it places constraints on the software. For example, one could define the state of a sector as either *fix* or *changing*. The *changing* state means that some aspect of the sector is undergoing a change, such as the sector boundary. Not only that, but the adjacent sectors would have to be in the *changing* state for some period of time where their boundaries are being recomputed, and so forth.

The notion that a change to either an FPA or sector must be atomic in nature is analogous to treating the FPA or sector as a shared resource. This would then mean that, from the time a dynamic change is requested on an FPA or sector until that change has been completed, the FPA or sector would be locked. Clearly, there are consequences for locking an object, because

it would also mean that there could be a lock on the non-geometric aspects associated with the object: for example, if (as a result of a dynamic change to a sector boundary) the state of a handoff is locked, because the sector is changing]. A relevant factor in this discussion is the amount of time that it would take to effect a dynamic FPA or sector change, because this would dictate the locking time on the object.

There is the possibility that when a change is initiated, such as assigning an FPA to a sector, the majority of the work associated with the change could be performed as a background process. During this processing, no other change would be permitted. Thus, when the state of a sector (or FPA) is changing, not only would the new geometry be computed (including possible changes to other sectors and/or FPAs, as noted above), but the data elements associated with the sector change would also be recomputed. For example, the elements of an inbound list to a sector may change as a result of combining two sectors. After all the changes have been prepared, the sector would be locked for a relatively short period of time, while the changes were displayed.

It is our understanding that the current host system provides procedural mechanisms to avoid possible problems about dynamic state data for a sector.

3 Requirements Specification

3.1 Scope

The FAA En Route system is changing with respect to how the system functions. Some of these changes are hardware in nature, while some are software in nature. To discuss the En Route evolution we define the following:

- HCS: host computer system
- PVD: plan view display
- DSR: display system replacement

Using the above notation, we present the En Route evolution paths in Figure 11.

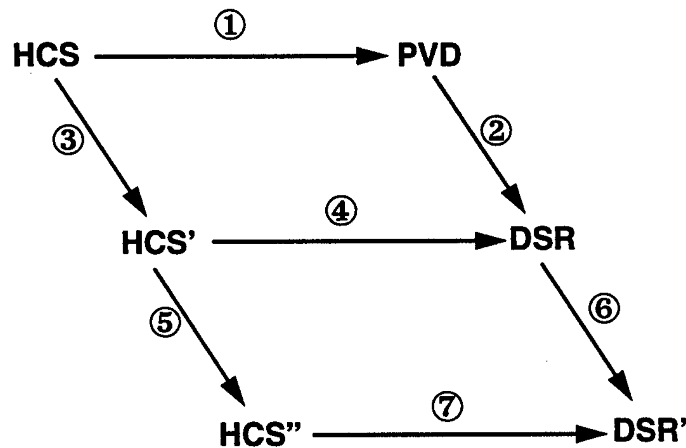


Figure 11: En Route Evolution Path

The above figure is explained in the following way:

- The currently deployed HCS interacts with PVDs, which provide the display capability. This interaction is denoted by the symbol ①.
- The PVDs are being replaced by DSR, which largely consists of new workstations and associated infrastructure, both hardware and software. The transition from PVDs to DSR is shown by the symbol ②.

- To accommodate the change to DSR, certain changes must be made to the HCS. This transition is shown by the symbol ③ and will result in a modified HCS, denoted by HCS'.
- After replacement of the PVDs by DSR and modifications to HCS, there will be a new system, whose interaction is shown by the symbol ④. This system is planned for installation in 1997.
- The HCS and/or modified HCS requires the ability to be upgraded. Hence, there is an intended migration of HCS to a new HCS, which is denoted HCS". The migration from HCS' to HCS" is shown by the symbol ⑤.
- Given an upgrade to the modified HCS, it will also be necessary to make some degree of change to DSR. This will result in a modified DSR, denoted DSR'. The evolution of DSR is denoted by the symbol ⑥.
- Finally, we have a final state where a new HCS interacts with a modified DSR. This interaction is indicated by the symbol ⑦.

The scope and degree of changes that will be made to the En Route System are significant. Among the changes that require consideration, we note the following two items:

- What is the appropriate choice of technology on which to base the new HCS?
- Should the new HCS be acquired as a COTS product or should the FAA go through a full-scale development, as was done for DSR?

These questions are complex and intertwined. The latter question is particularly relevant to the requirements that will be presented below. For example, if one accepts the premise that a COTS product(s) will be used for the HCS replacement, then one has the following options:

- Accept the COTS functionality as provided. This may, or may not, permit resectorization, for example.
- If the COTS product does not provide for some feature, pay to have that feature incorporated into the COTS product.

In the end, the role of FAA requirements for En Route functionality can significantly affect the acquisition strategy. In the case of this report, we assume that the requirements are developed in the context of a new HCS.

3.2 Assumptions

Clearly, a full specification of requirements for resectorization is beyond the scope of this report.We shall *not* address the following topics:

- initialization processing: This includes processing of adaptation data files.
- mode-specific requirements: The system can operate in multiple modes (which would include simulation for example), and this will not be considered here.
- shutdown processing: Certain functions are performed as part of system shutdown, but this processing will not be discussed.[1]
- flight strips: We do not address the routing and printing of flight strips.

In short, the requirements presented in this section are based on a steady-state model of the system.

3.3 Software Requirements

This section specifies that subset of the software requirements that are relevant to the problem of sector combination. The basic system functions that are considered here are shown in Figure 12. The labels in Figure 12 denote data transfer between the functions, with the arrows labeled by the nature of the data that is transferred.

In developing the requirements for the resectorization processing, we have used existing FAA documentation to the maximum extent possible.[2] However, our efforts in developing the requirements were limited by some issues with current FAA documentation. For example, we noted that there are a number of requirements that are either unspecified or implemented as a procedural manner. This point is discussed further in Section 3.4.

We believe that the requirements specified below represent a reasonable subset of what are appropriate requirements, subject to issues of the role of procedural requirements, discussed in Section 3.4.

We feel it is appropriate to make a general observation about the requirements documents that were available to us. That is, we believe that the way these documents are specified could be improved. For example, we would recommend the use of state transition diagrams. We would further recommend a more robust specification of state data. (For example, it is not really clear what state an FPA is.) A final consideration is the use of a formal specification approach to developing and presenting the requirements.

---

1. There are interesting questions about shutdown. For example, it is possible to cancel a system shutdown and such an activity could affect the state of information about sectorization.
2. Specifically, we have used the CD-ROM distribution of the PAMRI (Peripheral Adapter Module Replacement Item), version 1.3 distribution.

The development of easily understood and correctly specified requirements is crucial to the FAA, whether a component of the En Route system is contracted out or acquired from a vendor as part of a COTS acquisition.

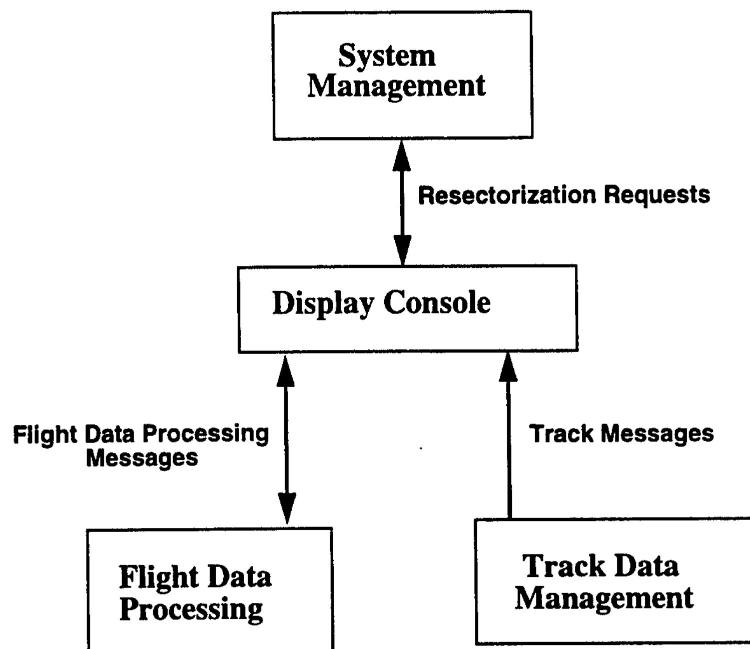


Figure 12: Simplified Functional Data-Flow Model

3.3.1 System Management

System management is responsible for the overall management of components within an En Route Center, such as consoles and interfaces to external systems. System management also helps in the detection of possible faults and is responsible for the reconfiguration of failed devices.

The basic functions performed include the following:

- system state data management (see Section 3.3.1.2)
- sector airspace management (see Section 3.3.1.3)
- display status information for sectors (see Appendix C)

3.3.1.1 General Requirements

The following general requirements apply to system management:

1. The system shall support multiple system management consoles.
2. Each system management console shall be capable of accepting any valid command at any time. The capability shall be provided to ensure that potential conflicts due to similar commands being entered from multiple consoles are handled in a consistent manner.
3. Each console shall not require the processing of a command to complete before accepting and initiating processing for another command.
4. Each console shall maintain consistent state data, as necessary.

3.3.1.2 State Data Management

The following subsections identify requirements for the management of state data for system management.

3.3.1.2.1 Component State Data Management

This section defines requirements associated with the state for components of an En Route Center. The term component is used here to denote a hardware component.[1]

3.3.1.2.1.1 Console State Data

The following general state data shall be maintained for each console:

- an identifier that uniquely identifies this console
- an physical address that may be used to communicate to the console
- the Console_Function, being either *Null*, *System_Management*, or *Radar_Display*

1. This can easily be generalized to include software components also.

3.3.1.2.1.2 Radar Console State Data

The following specific state data shall be maintained for each radar display console:

- information about the map current displayed, including
  - an identifier of the map
  - the center coordinates for this map
- beacon code list

3.3.1.3 Airspace Management

The airspace management function provides management capability for the En Route Center airspace (defined in terms of fix posting areas and sectors) including the ability to

- combine (and decombine) one or more sectors with (from) an existing sector
- assign a radar display console to another sector
- assign an FPA to a sector
- assign an FPA to another FPA

3.3.1.3.1 Airspace State Data Management

3.3.1.3.1.1 Node Data Management

The system shall provide for the management of nodes, where a node is defined as a point in space. The following state data is associated with a node:

- the name of the node, which is required to be specified as an alphanumeric string of at most six characters
- latitude
- longitude

3.3.1.3.2 Fix Posting Area State Data

The following state data shall be maintained for each fix posting area:

- an identifier that uniquely specifies the FPA
- the FPA *current_state*, being either *primary* or *subjugate*
- the FPA type, being either *sector*, *no_airspace*, *approach_control*, or *ARTS_approach_control*
- the function of the FPA, being either *FDEP*, *ARTS_III*, *FDEP_and_ARTS*, or *Adjacent_Center*

- the sector to which the FPA is currently assigned
- the sector to which this FPA is assigned in the initial state associated with the adapted sector plan in effect

3.3.1.3.3 Sector State Data

The system shall maintain the current sector plan that is in effect.

The following state data shall be maintained for each sector:

- an identifier that uniquely specifies the sector. (The identifier may be either a number or a name.)
- the current *sector_state*, being either *active* or *inactive*
- the *Sector_Use* (which is one of *Current_Training*, *Permanent_Training*, or *Operational)*
- the *Sector_Altitude_Type*, which may be one of the following: *high_altitude_sector*, *low_altitude_sector*, or *ARTS_ARTCC_Adjacent*
- the sector *CA_Boundary_Constant* (APSB), used for display of conflict alerts for the sector
- the sector *CA_MCI_Boundary_Constant* (APSC), used in conflict-alert processing and display eligibility of MCI alerts for the sector
- conflict-alert list (and associated state; see Section 3.3.2.3.3)
- inbound list (and associated state; see Section 3.3.2.3.1).
- hold list (and associated state; see Section 3.3.2.3.2)
- VFR inhibit list
- MCI function
- Group_Suppression_List associated with the sector
- an identifier for the printer associated with the sector
- an identifier of the console where the sector is currently displayed

3.3.1.3.4 Sector Plan Management

The capability shall be provided to apply an adapted sectorization plan to an En Route Center. This request may be made as part of system initialization or later, during system operation.

3.3.1.3.4.1 Acceptability

The following requirements apply to the acceptability of a request to apply a sector plan to a facility:

- The request must be entered from a system management console.
- The specified sector plan shall be valid.

If either of the above conditions exists, the request shall be rejected.

3.3.1.3.4.2 Processing

Upon acceptance of a valid request to apply a sectorization plan, the following requirements applies:

- For each sector whose assignment changes, a notification shall be sent to its corresponding display console(s) indicating the new assignment.

3.3.1.3.5 Sector Combination

The capability shall be provided to combine two or more sectors into a new sector.[1] This provides flexibility with respect to airspace control.

3.3.1.3.5.1 Acceptability Criteria

The functional form of a request to combine sectors may be written as follows:

$\{I_S\}$ --->C_S

where $\{I_S\}$ denotes a set of initial sectors and C_S denotes a controlling sector (i.e., the sector that will assume control of the sectors in the set $\{I_S\}$). The following requirements apply to a request to combine sectors:

- The request must be entered from a valid console as defined in the Input Message Eligibility Record adaptation data.
- All sectors shall be defined in adaptation data.
- There shall be at least one sector in the set $\{I_S\}$.

1. This is case (b) of the CS message in NAS-MD-311.

- Each sector in the set $\{I_S\}$ that is not identical to the controlling sector must be active.
- If there is a sector in the set $\{I_S\}$ that is identical to the controlling sector, it must have at least one FPA adapted to it in the adapted sector plan currently in effect.

If any of the above conditions exist, the request shall be rejected.

3.3.1.3.5.2 Processing

The following requirements apply to the processing of a valid request to combine two or more sectors.

If the controlling sector is identical to one of the sectors in the set $\{I_S\}$, all of the FPAs adapted to that sector in the adapted sector plan currently in effect shall be assigned to that sector plus any other FPA(s) currently assigned to that sector. When merged FPAs are affected by this action, the following applies:

- If a primary FPA is included in the new combination, its subjugate FPA remains merged with it.
- If a subjugate FPA is included in the new combination without its primary FPA, the merger is broken, and the primary FPA remains assigned according to current sectorization.

If the controlling sector is not one of the sectors in the set $\{I_S\}$, all FPAs, including merged FPAs, assigned to the specified sector in the current sectorization are reassigned to the controlling sector.

A notification shall be provided to each sector, including the designated controlling sector, indicating the intent to be combined into the sector designated as the controlling sector.

3.3.1.3.6 Assignment of a Sector to a Radar Display Console

The capability shall be provided to assign a radar display console in one sector to the radar display console position in another sector. This also includes the capability to reassign a radar console to its originally adapted position.[1]

1. This is case (c) of the CS message in NAS-MD-311.

3.3.1.3.6.1 Acceptability Criteria

The functional form of a request to assign a radar console in one sector to a radar console in another sector may be written as follows:

S_S--->S_D

where S_S is the source sector which will be assigned to the destination S_D.

The following requirements apply to a request to assign a radar display console in one sector to the radar display console position in another sector:

- The request must be entered from a system management console.
- When at least one request for radar-console assignment is in effect, the only other acceptable requests for that console that lead to a change in sectorization shall be another request for radar-console assignment.
- All sector numbers must be adapted.
- Each sector shall be currently assigned to a radar-console position.
- The sector S_S must be in an active state, and the sector S_D must be in an inactive state.
- If sector S_S is equal to S_D, the request is interpreted as a reassignment of radar console to its original position.
- No more than one radar console may be assigned to another radar-console position.

3.3.1.3.6.2 Processing

Upon acceptance of a valid request to assign a radar console to a specified sector, the following shall be performed:

- A notification shall be sent to each affected radar console initiating the requested change.

3.3.1.3.7 Assignment of an FPA to a Sector

The capability shall be provided to assign an FPA to a specified sector.[1]

1. This is case (d) of the CS message in NAS-MD-311.

3.3.1.3.7.1 Acceptability Criteria

The following requirements apply to this function:

- The specified FPA and sector shall be valid.
- The specified FPA must not be of type FDEP, ARTS III, combined FDEP/ARTS III, or associated with an adjacent center.

3.3.1.3.7.2 Processing

Upon acceptance of a valid request to combine an FPA with a sector, the following shall be performed:

- A notification shall be provided to the display consoles associated with the specified FPA and, if different, the display console associated with the specified sector.

3.3.1.3.8 Assignment of One FPA to Another FPA

The capability shall be provided to assign one FPA to another.[1] The FPA that is proposed to be assigned will be denoted as the *source FPA*, and the proposed receiving FPA will be denoted as the *destination FPA*.

3.3.1.3.8.1 Acceptability Criteria

The functional form of a request to assign one FPA to another FPA may be written as follows:

$F_S \dashrightarrow F_D$

where F_S is the source FPA that is to be assigned to the destination FPA, denoted F_D.

The following specifies the acceptability criteria for assigning one FPA to another FPA.[2] The following general requirements apply:

- Both the source and destination FPAs shall be one of the following types: *sector, "no air-space," approach control, or ARTS approach control.*
- The source FPA must not currently be a subjugate FPA.

---

1. This is case (e) of the CS message in NAS-MD-311.
2. See page 6-39 of MD311.

The following specifies the assignment of one FPA to another FPA:

- An FPA whose type is *sector* may be assigned only to another *sector* FPA.
- An FPA whose type is "*no airspace*" may be merged only with another "*no airspace*" FPA.
- An FPA whose type is *approach control* may be assigned to a *sector* FPA or to another *approach control* FPA.
- An FPA whose type is *ARTS approach control* may be assigned to a *sector* FPA or to another *approach control* FPA.

The following specifies the assignment of an FPA to itself:

- An FPA whose type is *sector* may not be assigned to itself.
- An FPA whose type is "*no airspace*" may not be assigned to itself.
- An FPA whose type is *approach control* may be assigned to itself only if it is currently a subjugate FPA.
- An FPA whose type is *ARTS approach control* may be assigned to itself only if it is currently a subjugate FPA.

If any of the above criteria are not satisfied, the request to combine FPAs shall not be accepted.

3.3.1.3.8.2 Processing

Upon recognition of a valid request to combine one FPA with another, the following shall be performed:

- For the appropriate FPA(s), a notification shall be sent to the display console that is currently controlling the airspace associated with the FPA.

3.3.2 Radar Display Console Processing

The Radar Display Console Management Function is responsible for managing the state of a display console. This includes interaction with the System Management Function to perform functionality appropriate to the local console (such as initialization) and operations on sector and/or FPA combination and decombination. This function is also responsible for displaying track data and lists, such as the inbound list or hold list.

3.3.2.1 Local Sector Airspace Management

Each display console shall be capable of providing support to locally manage its airspace. This results from interaction with the System Management function.

3.3.2.1.1 General Resectorization Processing

The following general requirements apply when the sectorization is changed:

- If a display console receives a notification to change the current sectorization and an error is detected, the request shall not be processed, and a notification shall be sent to the sender of the request.
- If a valid request to change the current sectorization is received, an acknowledgment shall be sent to the console that initiated the request.

The following specifies the processing to be performed when a valid notification to change the current sectorization is received. If, as a result of a resectorization request, one of the consoles transitions to an inactive state, the following processing shall be performed for that console:

- The console shall cease display of track and flight-plan data.
- The console shall not display the hold, inbound, or conflict-alert lists.
- The console shall not accept commands from a controller.
- The console state shall be set to *inactive*.

The following processing shall be performed for the console that will assume control of new airspace. Any state data associated with the airspace being combined shall be transitioned to the assigned airspace (and associated console) in the following manner:

- The assigned console shall receive and display the track display information for the console geometry.
- Flight-plan state data shall be updated to reflect the new airspace configuration.
- The hold, conflict-alert, and inbound lists shall be updated to reflect the new airspace configuration. That is, all data associated with the resectorization shall be transitioned to the new console. For example, elements of the hold list that are to be controlled by a new console shall be displayed on that console. Such data shall no longer be displayed on the other console(s).

When resectorization processing is complete, a notification shall be displayed on all consoles participating in the recombination indicating that the resectorization has been completed.

3.3.2.2 Display of Track Data

The system shall have the capability to display information associated with a track.[1] The information shall contain the following:

- a symbol denoting the current position of the track. Different symbols shall be used to denote one of the following: flight-plan aided track, free track, coast track, or a track currently in a hold state.
- a symbol denoting the track velocity vector

In addition, the following information shall be displayed:

- aircraft identification
- assigned altitude
- reported altitude
- computer ID number (CID)
- information about the track state (e.g., in handoff or hold), beacon code, and/or ground speed

3.3.2.3 Display Lists

The system shall provide the capability to display lists of information for the operator. These include inbound, hold, and conflict-alert lists. For each list, the controller shall have the ability to

- enable and disable the display of the list
- position the list at a particular location on the display console

Default values of the above items shall be specified in adaptation data.

3.3.2.3.1 Inbound List

The inbound list shall be subdivided into sublists, with each sublist denoted by the first posted fix within a center boundary.[2] The sublists will be presented in alphabetical order by the name of the first posted fix. For each sublist, the following information will be displayed:

---

1. This information is based on FAA specifications [FAA 95c](e.g., page 3-7). We refer to the full data block.
2. This information is based on FAA specifications [FAA 95c] (e.g., page 3-24).

- aircraft identification
- assigned altitude
- assigned beacon code

The entries in each sublist shall be ordered according to the center boundary crossing time, with the earliest time first.

3.3.2.3.2 Hold List

The hold list shall be subdivided into sublists, with each sublist denoted by the hold fix as specified in a hold message, first posted fix within a center boundary.[1] The sublists will be presented in alphabetical order by the name of the hold fix. For each sublist, the following information will be displayed:

- aircraft identification
- hold time
- aircraft altitude

The entries in each sublist shall be ordered according to decreasing aircraft altitude.

3.3.2.3.3 Conflict-Alert List

The conflict-alert list shall contain information about pairs of tracks for which a potential conflict alert exists.[2] For each pair, the following information shall be displayed:

- aircraft identification for each aircraft in the conflict pair
- identity of the sector that has control of the aircraft
- the center identity, if an adjacent center is exercising control of either aircraft

3.3.3 Flight-Plan Management

3.3.3.1 Management of Flight-Plan State Data

The following information constitutes the state data associated with a flight plan:

- Flight_Plan_ID, which uniquely identifies a flight plan

---

1. This information is based on FAA specifications [FAA 95c] (e.g., page 3-25).
2. This information is based on FAA specifications [FAA 95c] (e.g., page 3-27).

- state, which can be either active or inactive
- hold_state, which indicates if the flight is currently in a hold
- assigned_altitude
- aircraft_data
- beacon_code
- speed
- coordination_fix
- coordination_time
- requested_altitude
- route_information
- comment_text

3.3.3.2 Distribution of Flight-Plan Data

The capability shall be provided to distribute flight-plan data to one or more display consoles in a center. Such data shall be based upon receipt of any of the following messages:

- Activate flight plan
- Amend flight plan
- Assign flight plan altitude
- Assign flight plan beacon code
- Deactivate flight plan
- Delete flight plan
- Flight plan
- Hold flight plan

Upon receipt of any of the messages listed above, the following shall be performed:

- Relevant information shall be distributed to those display consoles that are holding information for a specific flight plan.
- The current state of a flight plan shall be maintained for use by other functions.

3.3.3.3 Flight-Plan Extrapolation

The capability shall be provided to extrapolate a specified flight plan. Flight plan extrapolation is used to determine[1]

- when flights inbound from an adjacent ARTCC are eligible for inclusion in an *Inbound_List*
- the next and previous sector-posted fix of the extrapolated route
- when a flight qualifies for exclusion from a *Hold_List* display
- when an extrapolated flight-plan position is past the first posted fix in a sector
- when an extrapolated flight-plan position is past the last posted fix in a sector

3.3.3.3.1 Inbound List Eligibility

A flight that is inbound from an adjacent ARTCC will be eligible for inclusion in the inbound list. Such a list is displayed at the sector position where the flight plan will enter the center airspace.

For all active inbound flight plans that are not eligible for display on an inbound list, the following check will be made:

$T_p \leq (BCT - ILET)$

where

- T_p is the current clock time.
- ILET is the inbound list eligibility time (in minutes) prior to the calculated boundary crossing time.
- BCT is the boundary crossing time for the inbound route segment.

The first time that the above inequality is true, the associated flight plan is eligible for display for the first controlling sector in the ARTCC.

1. See page 3-8 of NAS-MD-313.

3.3.4 Track Management

3.3.4.1 Management of Track State Data

The following state data shall be maintained for a track.[1] Track-state data shall be recovery protected in the case of a fault.

- **Track category**
  - operational
  - simulated
  - computer ID
- **Current track control**
  - location type
    - null
    - adjacent ARTS (automated radar terminal system)
    - adjacent NAS (National Airspace System)
    - this center
    - external ARTS
    - interfacility point-out status
  - location identifier
  - computer ID value
- **Track position**
  - X-position
  - Y-position
  - altitude
  - assigned
    - type (none, single altitude, OTP altitude, blocked altitude, AFA altitude, VFR altitude)
    - value
  - reported
    - data source (mode C or controller entered)
    - value

1. The information is based on pages 1-23 of FAA specifications [FAA 95h].

- interim
    - available (boolean)
    - value
- **Track velocity**
    - X component
    - Y component
    - ground speed
- **Associated flight-plan ID**
- **Beacon code**
    - established (boolean)
    - beacon code has changed (boolean)
    - current value
- **Conflict information**
    - conflict indicator (boolean)
    - controlling console
- **Handoff information**
    - handoff to/from other location
    - handoff type
        - to adjacent ARTS
        - to adjacent NAS
        - adjacent ARTS to this center
        - adjacent NAS to this center
    - location ID
    - handoff within this center
        - current controlling sector
        - receiving sector
    - auto-handoff state
    - inhibited
        - manually
        - by program
        - by program, temporarily
    - not inhibited

- **Controlling track sector history data**[1]
  - fourth oldest controlling sector
  - third oldest controlling sector
  - second oldest controlling sector
  - first oldest controlling sector
- **Track status indicators**[2]
  - emergency (beacon code 7700 correlated)
  - radio failure (beacon code 7600 correlated)
  - OLD (Crosstell track data timed out)
  - FAIL (DR received for TI or TA)
  - HDFH (H handoff)
  - HDFO (O handoff)
  - HDFK (K handoff)
  - HOLD (hold at present position)
  - CST (coast)
  - SBCO (special code 1 correlated)
  - SBC1 (special code 2 correlated)
  - RCVB (received beacon code used for smoothing not equal to assigned beacon code)
  - NONE (no beacon code received but code is assigned)
  - BLNK (blank)
  - GSPD (ground speed of flat tracked aircraft to be displayed)
  - MSAW (E-MSAW alert)
  - MOFF (E-MSAW specific suppress)
  - MIFF (E-MSAW indefinite suppress)
  - track is an ARTS arrival
  - track detected across a sector boundary without transfer of control

---

1. This is called sector history data, but what is actually stored is a pointer to a table that contains addresses of consoles.

2. This information is mainly referenced for E-field display.

3.3.4.2 Distribution of Track Data

The capability shall be provided to distribute track data to radar-display consoles. These data are then used for display and operator management.

When information about either a new track or an update to an existing track is received, the following requirements apply:

- The information shall be distributed to those display consoles that are currently displaying the track.
- Current information about all tracks shall be maintained for use by other functions.

3.3.5 System Capacity Requirements

The following table lists parameters that define system capacities and design requirements. It is a subset of the requirements presented in Annex A of FAA specifications [FAA 95f]. The values below are intended to apply to every ARTCC.

Parameter	Value
Tracks	700
FPA boundaries	255
Total FPAs	400
Total approach control and ARTS approach control FPAs	255
Total fixes	6000
Sectors	100
Adjacent centers	10
Number of track update messages per subcycle	200
Number of active flight plans	2500

Table 1: System Capacity Requirements

3.3.6 System Timing Requirements

The following table lists system timing requirements. The values are largely taken from FAA specifications [FAA 95e].

Parameter	Value
Start-up	2 min
Startover	10 seconds
Switchover	10 seconds
NAS-to-Glass	1 second

Table 2: System Timing Requirements

3.4 Procedural Requirements

There are a number of procedural requirements that apply to the combination of a sector and/ or FPA with another sector and/or FPA. These requirements are listed below, based on our current understanding of procedural semantics within an ARTCC. After a presentation of each requirement, we provide a comment regarding the degree to which such a requirement could be implemented in software.

1. Two (or more) sectors can be combined only if they each contain an FPA that shares a common boundary with the other.
   Comment: This would require that the proposed sector(s) for combination be examined to make sure that there is a shared common boundary.

2. Two (or more) FPAs can be combined only if each contains a common boundary with the other.
   Comment: This would require that the proposed FPA(s) for combination share a common boundary.

3. Two (or more) sectors can be combined only if they are within the same operational area. Two (or more) FPAs can be combined only if they are in the same operational area. An *operational area* is a set of sectors within which an operator is certified to control aircraft.
   Comment: This could be implemented in software by defining an operational area in adaptation data, either in terms of sectors or FPA(s). Then, for a proposed sector and/or

FPA combination, a check would be necessary to make sure that the elements are in the same operational area. Another alternative would be to associate an operational area with an FPA, then verify the equality of operational areas when a sector and/or FPA combination is proposed.

4 CORBA Approach

4.1 CORBA

4.1.1 Background

CORBA, the *Common Object Request Broker Architecture,* is an industry standard developed by the Object Management Group (OMG). Within OMG, there are approximately 500 members from industry, universities, and government labs; the OMG is often cited as one of the largest software consortia.[1] The cost to join OMG is a function of a member's activity. For example, industry costs can be substantial, depending upon the sales of the company, while an academic membership is relatively inexpensive. However, there is a direct relation between the amount of financial contribution and the amount of influence a member has regarding what specifications are developed and approved by the OMG.

The main goal of the OMG is to promote the use of object technologies. This is achieved primarily through the development of specifications, such as CORBA [OMG 95a]. Another document particularly relevant to this report is the specification of *CORBA services* [OMG 95b].

The simplified process by which the OMG accepts a specification is outlined below:

1. A group of people recognize a need for a document specifying some functionality that is related to CORBA.
2. The group is granted a charter by the OMG to carry out the work.
3. The group develops a request for proposal (RFP). After the request is approved within OMG, it is publicized.
4. Vendors respond to the RFP stating the technology that they intend to implement. A vendor must also commit to having a commercial implementation within one year.
5. If multiple vendors submit proposals, they are encouraged to work together to seek a common solution.

1. The OMG Web site is http://www.omg.org/.

When the consensus is reached by the vendor(s), a vote is taken within OMG to accept or reject the proposal.

4.1.2 Architectural Overview

The architectural context for CORBA is a fundamental aspect of the specification and the way in which CORBA is used in a system. It is perhaps easiest to describe this as an interaction between a client and a target object. The client makes a request of the object, and a response is provided, as shown in Figure 13.

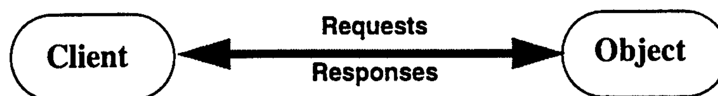


Figure 13: Interaction of Client and an Object

Figure 13 can be interpreted in terms of a client-server model. The mechanism used to achieve interaction between a client and an object is the *remote procedure call* mechanism.

One of the strengths of CORBA is to define an architectural model for the interaction of a client and an object. This context is presented in Figure 14.

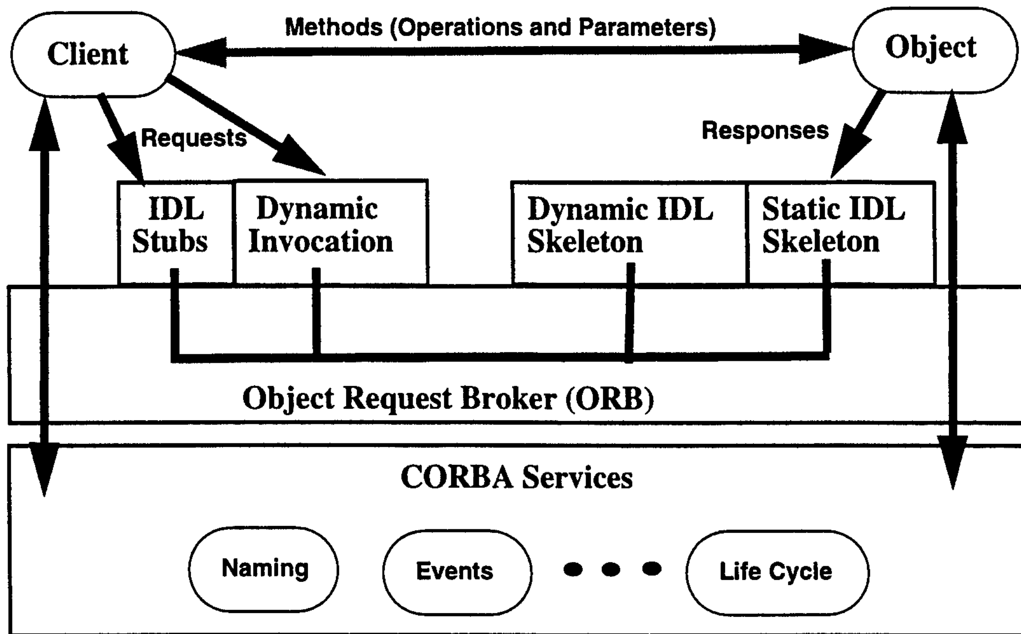


Figure 14: CORBA Architectural Model

The essential features of the architectural context include the following:

- A client makes a *request* of an object by invoking a method on the object. The *request* is invoked on an object and may have input and output parameters.
- An *interface description language* (IDL) is used to specify the interface between objects. From the IDL, client stubs are generated. The specification of the object is achieved by filling in the details of the IDL skeleton. The IDL allows for platform and programming-language independence and is a fundamental aspect of CORBA.
- The *object request broker (ORB)* is the basic arbitrator of interaction with an object. Among other things, the ORB provides for creation, activation, and communication between objects and other object management services.
- A number of CORBA services are available to provide support for naming, event management, and so on. These services are modeled on the notion of an object, and an application can invoke requests for a particular service, of which there are many. These services are described further in Section 4.1.5.

The discussion associated with Figure 14 was presented at a simple level to convey the basic elements of the CORBA architectural context. For more details, refer to the CORBA documentation.

An additional advantage provided by CORBA is flexibility. For example, if a new service is defined, it is specified in IDL, which is neutral with respect to language mappings. That is, a C or Ada language mapping can be defined for the new service. Hence, the IDL provides a level of abstraction that permits the development of flexible language bindings.

4.1.3 Real-Time Considerations

The current CORBA specification [OMG 95a] was not developed with a focus on the real-time domain. Within the past two years, a Real-Time Platform Special Interest Group (RTPSIG) has been formed. That group is developing RFPs that would lead to the development of a real-time CORBA. An RFP has been developed to include the following:

- real-time protocol (RT ESIOP)
- fixed priority scheduling
- pluggable transport layer

There is also an RFP for a minimal CORBA, which is intended to be a version of CORBA that is lightweight in its memory footprint, although not necessarily focused on real-time aspects.

4.1.4 Communication Mechanisms

CORBA provides a number of mechanisms for communication between objects. The basic interaction is through the use of a *method* invocation, which is an abstraction of a remote procedure call. A method is the code that is executed to perform a service of the object. If a client makes a request of an object, the corresponding method will be called on the object. The request (and method) specifies the name of the operation and parameters, both input and output. The delivery semantics of a method are reliable. It is also possible to specify additional, vendor-defined information, that can provide quality-of-service information. This is defined as a *context,* and the information is passed as a character string.

It is also possible to specify that a method be *one-way,* indicating that the caller will not be blocked. This form of interaction is subject to the following constraints on the caller:

- The delivery semantics are best effort and not guaranteed.
- There can be no output parameters, and the return type must be void.
- No application-defined exceptions can be raised, although predefined exceptions may be raised.

In a sense, the one-way method invocation is a form of asynchronous interaction in that the caller is not blocked. Although the above constraints are rather restrictive, the one-way method invocation is sometimes useful in practice. If output parameters were desired, a sender-

receiver model using one-way semantics could be developed. However, in this case, the application must perform more management activity, such as matching requests and responses. In this case, it becomes increasingly like a message-passing system rather than an object-interaction system. Another approach would be through the use of a callback mechanism.

CORBA also supports a *one-to-any* model (publish-subscribe), which is based on the notion of an *event*. Basically, there can be producers of events and consumers of events. The communication takes place over an event channel. This is defined in the Event Service specification [OMG 95b].

The manner in which a client and server interact can be specified in one of the following ways:

- static IDL: All the methods are known in advance to both the client and server. This is the most common form of interaction and is simple and efficient. It also provides for type-safe operation.
- dynamic IDL: The methods are defined at runtime. This provides more flexibility than the static IDL approach, but is believed to be more complicated and less efficient.

In this report, we will be mainly concerned with the use of static IDL specifications.

As a result of executing the method, a result may be returned or an exception may be raised. Exceptions may be of two types: those that are predefined by CORBA and those that are defined by an application in the IDL specification. As an example of those that are predefined by CORBA, if a method is invoked and a parameter is invalid, a predefined exception will be raised indicating that the parameter is invalid.

4.1.5 Additional Services

In addition to the basic CORBA architecture, there are a number of associated services defined that are available to an application. These are specified in OMG specifications [OMG 95b] and include the following:

- naming service: This service provides interfaces to define a naming context (a scope in which all names are unique) and binding for a name to an object.
- event service: This service provides interfaces for producers and consumers to communicate information about events. The communication between the producer and consumer is asynchronous in nature, as it is mediated by an event channel. Event distribution is reliable in nature. This is an example of a publish-subscribe communication model.
- life-cycle service: This service provides interfaces for creating, deleting, copying, and moving objects. It is also possible to perform operations on a graph of objects. An important concept in this area is the role of a *factory* (an object that creates other objects).

- persistent object service: This service provides interfaces for retaining and managing the persistent state of objects. Included is the ability to specify a persistent object, which has an associated identifier. Clients can then interact with the persistent object to manipulate state data.
- transaction service: This service provides interfaces for transactions. Two models are available: a flat transaction model and a nested transaction model, with the latter being optional. The service supports management of both system and application transactions. The service permits coexistence of a procedure-oriented paradigm with an object-oriented paradigm. Multiple transactions can be executed concurrently.
- concurrency control service: This service provides interfaces for a coordination mechanism among multiple clients for access to a shared resource. The coordination is achieved through the use of locks. For example, a client must obtain a lock before being granted access to the resource. This prevents multiple clients from performing a conflicting action on a shared resource. Read and write locks are among the locking modes supported.
- relationship service: This service allows for the specification of entities and relationships between them, where an entity is a CORBA object. For example, a flight plan *contains* a route; correspondingly, a route is *contained in* a flight plan.
- externalization service: This service provides the capability to record the state of an object in some storage media (e.g., memory or file system). The service also includes protocols and conventions for externalization (and the converse operation of internalizing) an object's state.

The existence of services, such as those described above, illustrates the philosophy in the development of CORBA by the OMG. That is, there is a basic architectural model that permits the specification of objects (using IDL, for example) and mechanisms for their interaction. The services supplement that basic model and allow for applications to make use of the services needed for a particular application context.

It is important to note that CORBA does not specify a particular operating system. There are, however, certain expected functions of an operating system that have been implicitly and abstractly specified.

4.2 Presentation of Design Information

For the purposes of this report, we will use a design notation that allows us to describe

- object classes
- objects as instances of a class
- methods available on an object
- event channels

An example of the notation for the use of CORBA methods is presented in Figure 15.

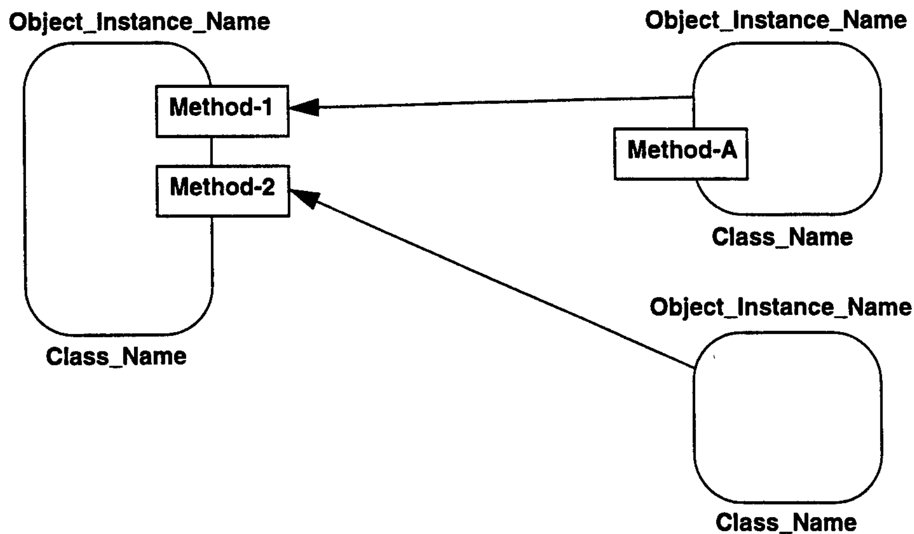


Figure 15: Notation for Use of CORBA Methods

The essential points about the above diagram are summarized below:

- An object class is represented by a rounded rectangle. The class of the object is listed beneath the object. If an instance of an object must be uniquely denoted, that name appears above the object.
- The methods that are provided by an object appear on the side of an object (class), and the name of the method is present.
- Invocations on an object are shown by a solid line with the tail of the arrow denoting the caller.

The intent of the above notation is to achieve a simple approach that can be easily understood. We recognize that object classes do not interact *per se*. The semantics of object interaction are *defined* in the context of a class and *implemented* through an instance of the class.

CORBA also provides for the use of event channels that permit an asynchronous interaction between the producer and consumer of the event. The four cases for how event channels can be used are illustrated in Figure 16.

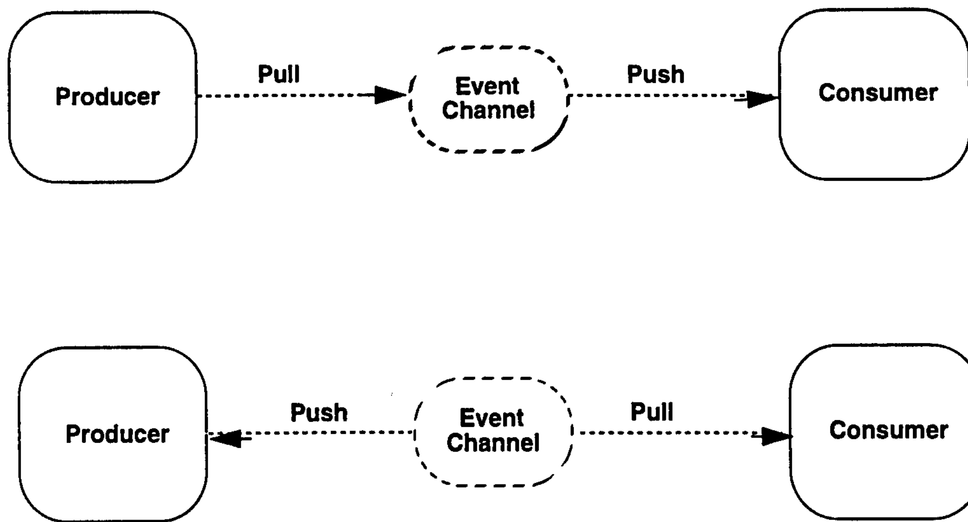


Figure 16: Notation for CORBA Event Channels

Figure 16 shows two examples of the notation for CORBA event channels. The event channel that is being used by a producer and consumer is labelled and appears as a dashed line. There are different ways in which a producer and consumer interact with the event channel. For example, the top of Figure 16 shows a pull producer and a push consumer; the bottom of Figure 16 shows a push producer and a pull consumer. The concept of push and pull denotes the way in which events are provided and consumed from the event channel. The ability to provide these different mechanisms allows versatility in how events are handled.

4.3 Basic Design Issues

Before discussing the design considerations specific to the resectorization problem within an ARTCC, we will discuss general design principles appropriate for designing systems using CORBA. Although CORBA promotes communication between objects and, thus, object-oriented design, it is not strictly necessary to create an object-oriented design to use CORBA; it would be possible to create a functional design and use CORBA to communicate between subsystems. However for the purposes of this report, we have considered an object-oriented design method with the expectation of exploiting all of CORBA's capabilities.

The design process is composed of a number of phases: object identification, interface determination, and design transformation. The key difference between object-oriented and more

traditional design approaches is that the former is based on data, while the latter is based on actions.

4.3.1 Object Identification

One of the hardest problems in object-oriented design is identifying the objects to be implemented. Perhaps the root cause of this difficulty is answering the question, "What is an object?" One answer to this question is that an object is an encapsulation of data with an interface that permits the data to be modified or recalled as needed by a user of the object. The important part of the answer is that an object *encapsulates* the data so that the only way the data may be accessed is through the interface similar to an abstract data type. With this in mind, we can now consider how to identify objects to be implemented.

If our design structure is going to last, given that there will be changes in requirements, it is important to choose the objects carefully. Objects should model things (for want of a better word) in the real-world system that are persistent in nature and whose interface will not change (or will vary in only a minor way) for their lifetime (although the implementation of the interface may change). Typically, this means choosing objects to represent physical objects with invariant structures and functions. A good example in our context would be an aircraft. Although the details of an aircraft will vary considerably over the life of the system, we expect that all aircraft will have a consistent set of attributes (X and Y coordinates, altitude, velocity, etc.). Of course, the things we model do not have to be physical in the sense of being solid; the things might be more ephemeral, such as tracks produced by an analysis of radar data. Again, there are certain attributes that we expect all tracks to have, and we expect that our system will always rely on the concept of a track for correct operation. Finally, the things that we model might not have any physical existence at all; we might choose to model a relationship between objects as an object in its own right. An example of this would be an object that models the relationship between a track and a flight plan. The consistent idea behind these examples (and suggestions as to what objects should be chosen) is that the objects should encapsulate certain data items and are expected to remain fairly constant throughout the lifetime of the system. More importantly, the data items encapsulated within an object should have a strong relationship to each other and not need a strong relationship to data items encapsulated in other objects.

4.3.2 Interface Determination

During the object identification phase, we will have some idea of how the software system is expected to interact with each object. We will have an intuitive idea of the relationships between different objects (either of the same or different types) and the needs that each object has of other objects, so that the object may perform its function correctly. For example, the

display of a track object (more precisely, an object representing a track) will need the coordinates of the track so that a determination can be made of the screen region that corresponds to those same coordinates. This example shows that the display object needs to access the track object's X and Y coordinates.

During the interface determination phase, the designer chooses the precise nature of the interfaces that the object will provide. In the example above, it would be possible for two interface routines to be provided: one to provide the X coordinate and one to provide the Y coordinate. Of course, we would not do this as we cannot imagine that any part of the system might want one coordinate without the other, and even if we did find that need, the burden of getting both co-ordinates and then "forgetting" the unneeded coordinate is small. We would generally choose an interface routine that provided both coordinates. But, what about other information that might be encapsulated in the object? Should the interface routine return altitude? Should it return velocity? The choices here are endless, and the decisions should be shaped by the ways that other objects will be permitted to modify (through an interface routine) or access the encapsulated data.

One of the advantages of object-oriented design is that new interface routines can always be added without disturbing existing interface routines (assuming that the new interfaces do not interfere with the function of the existing interfaces). Some objects may end up with many interface routines; this is typical of many object-oriented designs and may cause system implementors to have difficulty determining which interface routine to use.

Generally when adding a new interface routine, the designer should examine the existing routines and see if a generalization of one of the existing routines might serve the same purpose as adding the new routine. Of course, if the generalization requires a change in the syntax of the interface routine, all objects currently using the routine will have to be updated appropriately.

4.3.3 Design Transformation

The initial development of an object-oriented architecture can be performed in the context of a generic object system. Part of the design transformation is to refine that initial design in the context of a particular object system, in our case CORBA.

At this stage, particularly in the context of an ARTCC, it is likely that the design will consist of a number of objects, with the expectation that there may be *many* instances of each object. Realistically, particularly in the distributed object (CORBA) context, it takes time to communicate from one object instance to another, and as the number of instances increases, the system will be swamped by communication delay. The existing design should, therefore, be modified and collections of objects brought together to create the CORBA objects. The rules guiding this collation process are similar to those for identifying objects in the first place —

collect into one CORBA object the object instances that encapsulate closely coupled data items. For example, we might create an aircraft-manager object as a collection of all aircraft-object instances. As we go through this process, some of the interface routines may vary. We might (within the CORBA object) still treat the object instances as separate instances. However, we would not make their interface routines directly available for use by other CORBA objects.

If there are other CORBA objects that need the data for one of the hidden objects, we would need to add a new interface routine to the object container (in our example, the aircraft manager), and this routine would access the hidden object. There is a danger here, though, that we might not have gained much by the collation. If, as in our example, we hide the aircraft instances within the aircraft-manager object, we still had a need for some other object (such as the display) to access individual aircraft data, and the communication overhead still exists. The purpose of the collation and instance hiding is to minimize communications. Thus, the data collections should be treated *en masse;* otherwise the communications to the individual data items will become a bottleneck in system performance.

4.4 Architectural Considerations

4.4.1 Chosen Architecture

The architecture chosen for the CORBA aspect of this report will contain the following components:

- display consoles
- system-management consoles
- flight-data processing
- track-data management

The flight-data processing and track-management functions are intended to replace the current Host computer system processing. The incorporation of CORBA in the ARTCC architecture is based on the following:

Design Consideration C-1 DSR consoles will be allowed to contain CORBA-based objects.

To accommodate the above design consideration, we need the following, somewhat stronger statement:

> **Design Consideration C-2**
> Any component of the ARTCC, such as flight-data processing, track management, or system management, will be allowed to contain CORBA-based objects.

The resulting architecture, which includes the above design considerations, appears in Figure 17.

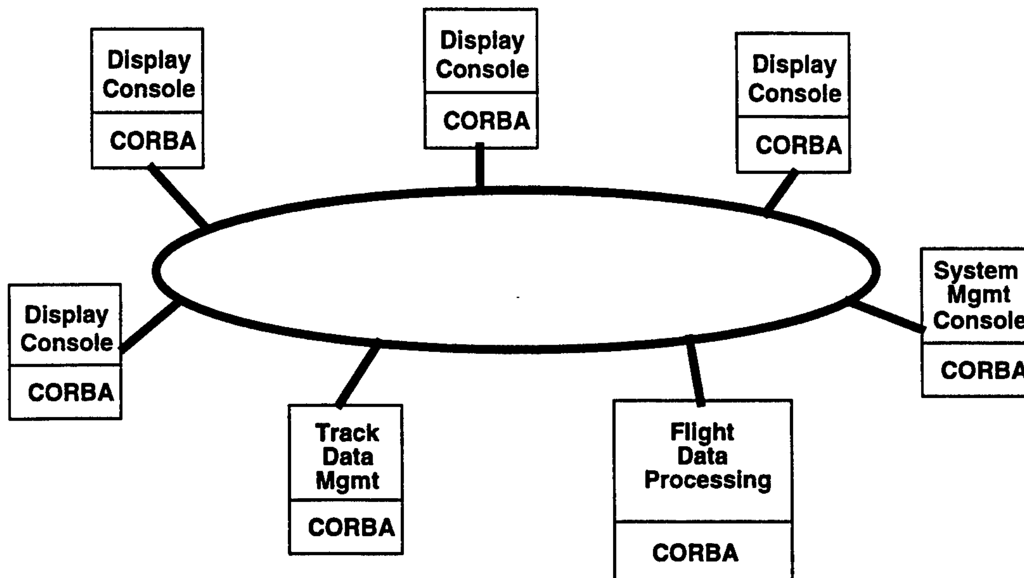


Figure 17: Assumed CORBA Architecture

The important point to note in Figure 17 is that we are assuming that CORBA is present on all system components. This would require a change to the current display system replacement (DSR) to accommodate a CORBA implementation. If CORBA is to be seriously considered for inclusion within an ARTCC in a long-term solution, it should be permitted to be fully distributed over all components, as needed.

The ring connectivity in Figure 17 should be interpreted as logical connectivity. We are aware that the DSR connectivity among consoles is a multi-ring structure, and there is nothing that would prevent maintaining that connectivity in the context of a redeveloped host. In this report, we emphasize the components rather than their physical connectivity.

4.4.2 Migration Considerations

The architecture described above may require many changes to the current DSR system. It is worth a moment to reflect on a possible migration path that would first permit the introduction of CORBA as part of redeveloping the host computer system. This could then be followed by a fully distributed CORBA implementation. An architecture to accommodate this migration approach is presented in Figure 18.

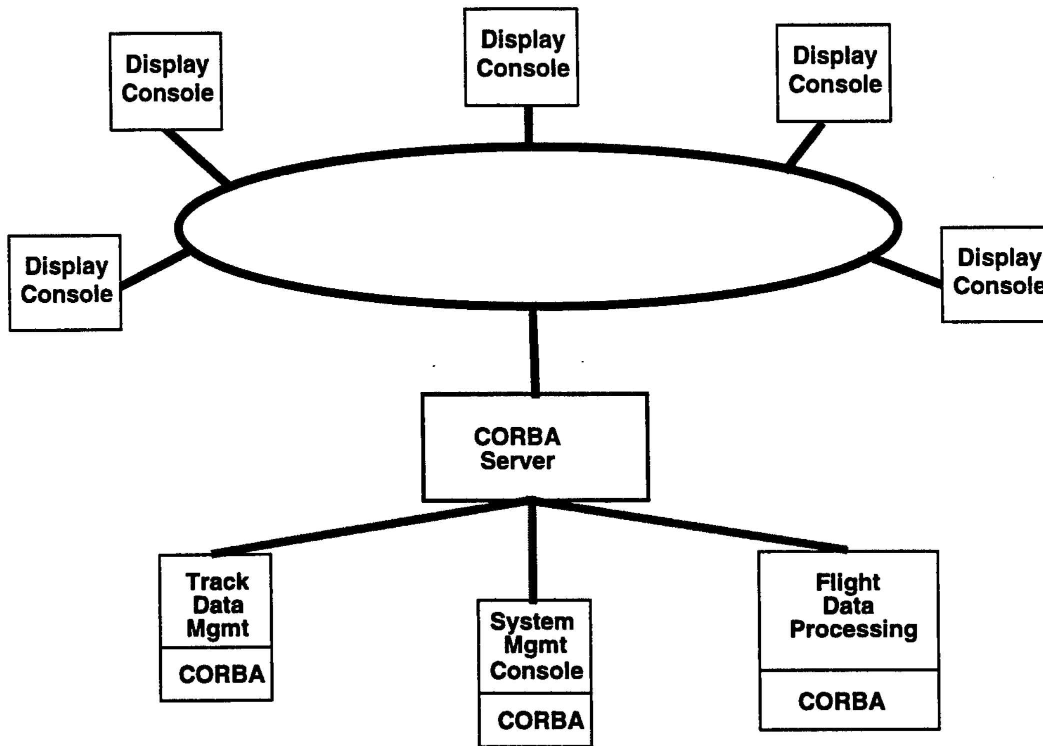


Figure 18: CORBA-Based Migration Architecture

The essential features of the above intermediate architecture are that it provides a mechanism to incorporate CORBA into the currently planned ARTCC. This is accomplished through the use of a CORBA server. In the above figure, system management and the new host functions (track-data management and flight-data processing) communicate to a given console by the CORBA server acting as an intermediary. Then, the CORBA server could communicate with the display consoles using the current DSR communication protocols, for example.

The architecture in Figure 18 is functionally equivalent to that shown in Figure 17. That is, in Figure 18, the objects are physically collocated (for ease of migration), whereas the objects are

physically distributed in Figure 17. Thus, a fully distributed CORBA architecture can be obtained by distributing the object functionality from the server onto the physical consoles.

Although the two architectural diagrams are functionally equivalent, there are several important non-functional differences. These include the following:

- The CORBA server would be a single point of failure. It would be necessary to incorporate fault-tolerant techniques to eliminate the risk of a single component failure.
- Performance could degrade with the server. The server would, in effect, have to manage multiple resources (objects), and this could lead to first in, first out (FIFO) queuing of method invocations. For example, the time to distribute track data would increase because the server is functioning in part as a protocol translator. Such additional processing comes at a price.
- The server could become complex. Because the server would have to communicate both with CORBA implementations and the current DSR protocols, not to mention the possibility of other components (e.g., CTAS or URET), it could quickly become a complex design.

4.5 Initial Design

In this section, we present the initial CORBA design for this work. The overall guiding principle in the development of the initial design is expressed as follows:

Design Consideration C-3
The initial CORBA design will be based on the maximal use of objects, without regard to implementation considerations.

This consideration permits us to concentrate on the functionality that is required without considering implementation details. This also implies that we will not consider, as part of the initial design phase, possible performance or fault-tolerant considerations. Such considerations could be treated as a refinement of the initial design. The implementation of the initial design will be developed in the context of a virtual machine.

The following is a list of the candidate object classes we will consider for the CORBA design:

- system management
- airspace management
- FPAs
- sector
- console

- inbound list
- hold list
- conflict-alert list
- track management
- track
- flight-plan management
- flight plan

Much of the information listed above is associated with the functionality of the display console. Figure 19 should help illustrate the relations among the relevant objects.

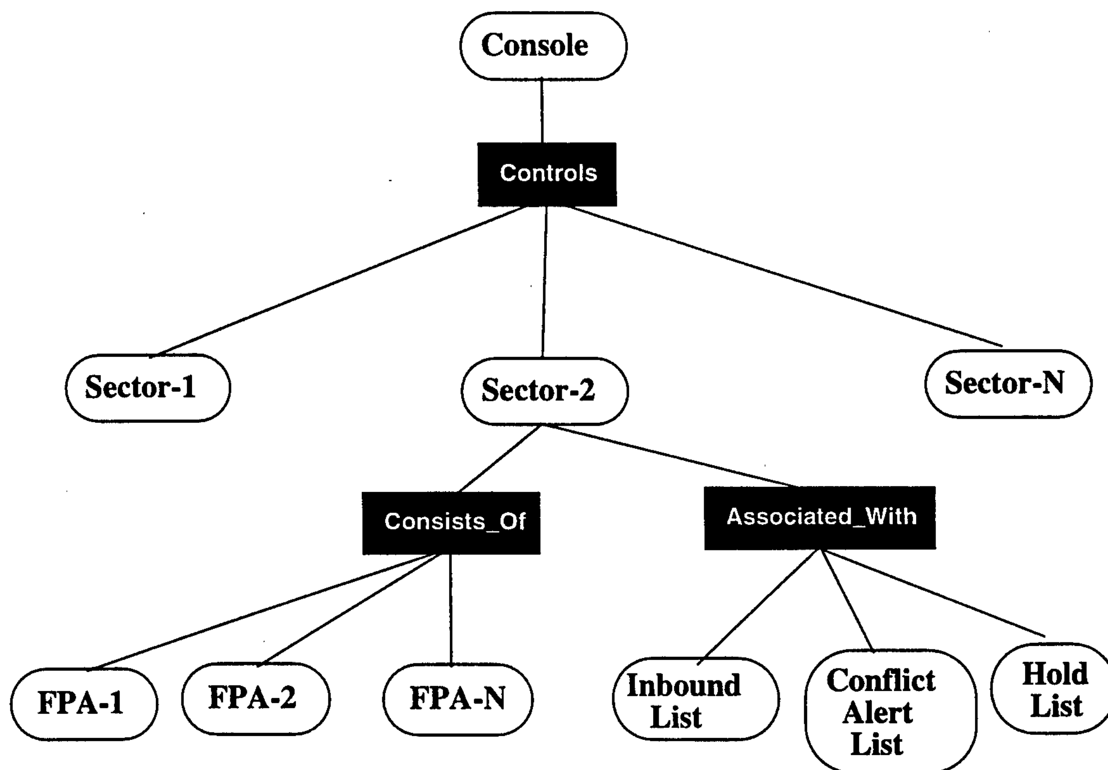


Figure 19: Console-Object Relationships

In the following subsections, we present a discussion of the candidate classes with emphasis on the state data and methods associated with each class. A brief discussion of descriptive information appropriate for the initial design is contained in Appendix G.

4.5.1 System Management

The purpose of system management is to maintain a list of all console objects, sectors, and FPA references for display and control.

The state data associated with system management includes the following:

- console object references: a database of system console object references
- FPA object references: a database of FPA object references
- sector object references: a database of sector object references

The methods associated with system management include the following:

- initialize: performs initialization for system management[1]
- console notify: provides the object reference of a console
- sector notify: contains the object reference of a sector that has been created, deleted, or modified
- FPA notify: contains the object reference of an FPA that has been modified

In terms of our notation, we represent the system management as shown in Figure 20.

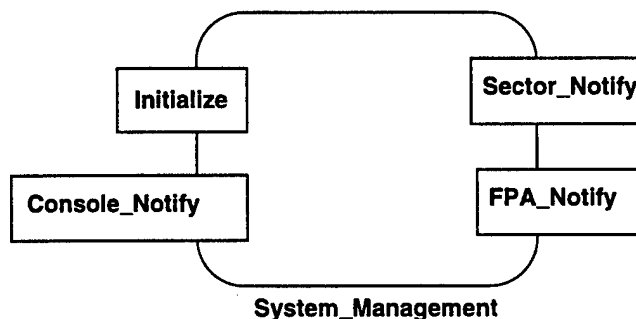


Figure 20: Initial Object Specification for System Management

1. Each object will have a method for initialization. For example, the invocation of the initialization method on an object may perform memory-management operations for local state data.

4.5.2 Airspace Management

The purpose of airspace management is to manage the creation, modification, and deletion of sectors. This function also distributes data to sector objects based on the sector and console geometry. The distribution of track data to this object is performed by the track-management object. As part of system initialization, all display and management consoles must register with airspace management.

The state data associated with the airspace-manager object includes the following:

- object references for all sectors in the system
- a database of FPA object references
- database of object references for all registered display consoles, with the console geometry and assigned sector
- database of references for all registered system management consoles

The methods associated with the airspace-manager class include the following:

- Initialize: performs initialization for the airspace-manager object
- Get available consoles: returns a list of object references for all known consoles
- Register display console: allows for a console to register with airspace management
- Register management console: allows for a system-management console to register with airspace management
- Create FPA: creates an FPA having a specified geometry and type
- Get FPA references: returns a list of object references for all the FPAs in the system
- Create sector: creates a sector object from a specified set of FPAs. once the sector object is created, the airspace manager updates the sector object with all relevant data.
- Delete sector: deletes a specified sector object and deassigns all FPAs associated with that sector
- Get all sector references: returns a list of object references for all sectors in the system
- Combine sectors: combines two (or more) sectors into one sector and returns an object reference for the combined sector
  - Split sector: removes a previously established sector combination (i.e., to split a sector into its constituent parts)
  - Assign sector: assigns a sector to a console and notifies the sector object of the assignment
  - Deassign sector: removes the assignment of a sector to a particular console
  - Track notify: provides notification that data are available for a new track or the data for an existing track have been updated

- Drop track: track object reference that has been dropped
- Purge tracks: all track object references dropped
- Flight plan notify: provides notification that either data are available for a new flight plan or the data for an existing flight plan have been updated
- Delete flight plan: indicates that the flight plan has been deleted
- Purge flight plans: indicates that all flight plans have been removed from the system
- Conflict notify: provides notification of a new conflict or updated information about an existing conflict
- Conflict resolved: indicates that the conflict no longer exists
- Console geometry changed: provides notification of a change in the geometry of a console

In terms of our notation, we represent the airspace manager object as shown in Figure 21:

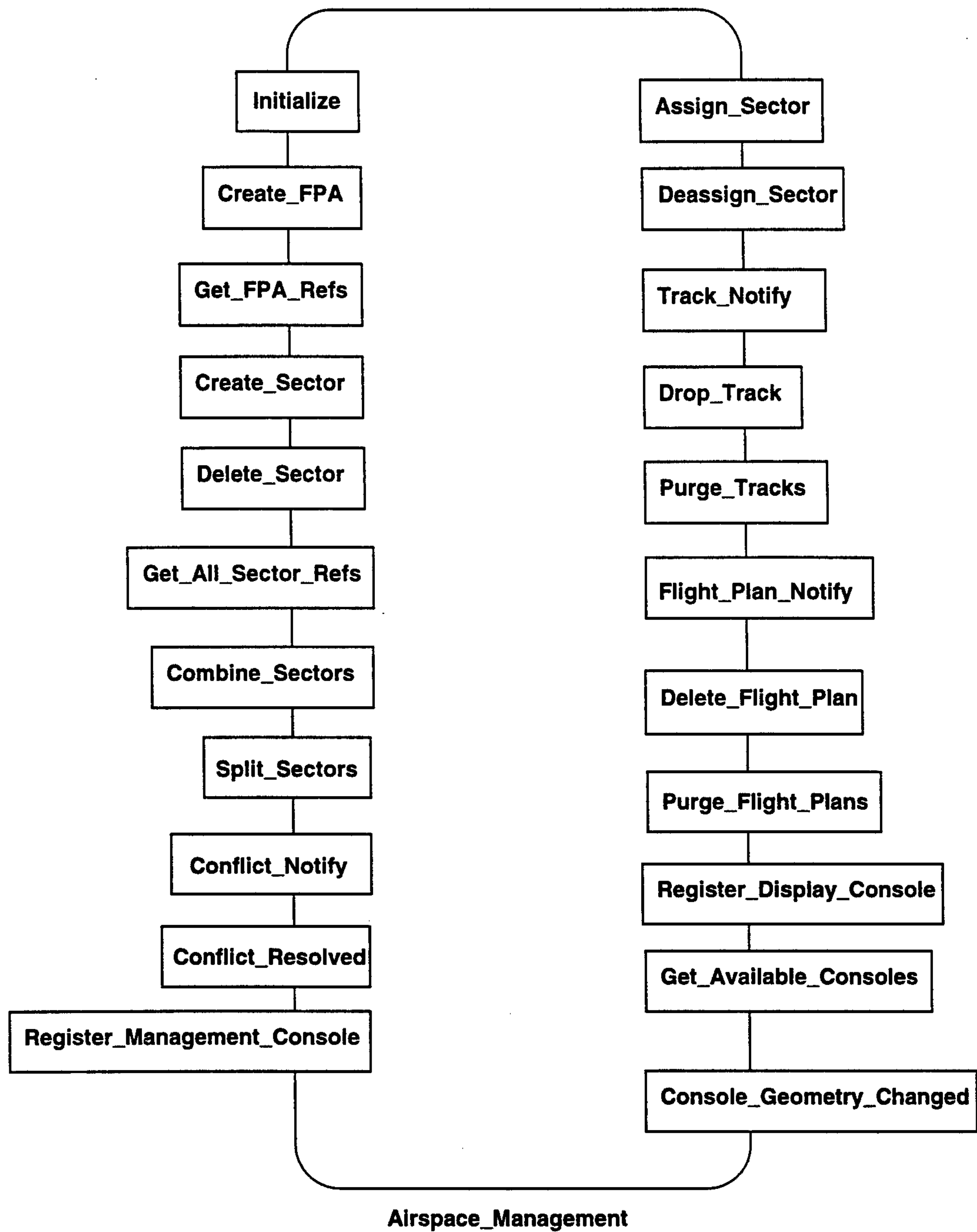


Figure 21: Initial Object Specification for Airspace Management

4.5.3 FPAs

As noted earlier in this report, the FPA is the fundamental unit of airspace within an En Route Center. We define an FPA class from an object-oriented perspective.

The state data associated with an FPA includes

- geometry
- type (such as controlled or no_airspace)
- default sector to which the FPA is assigned
- current sector to which the FPA is assigned
- a list of other FPAs that are assigned to this FPA
- the FPA to which this FPA is assigned

The methods associated with the FPA class include the following:

- Initialize: performs initialization for the object
- Get geometry: returns the geometry of a specified FPA
- Assign FPA to sector: assigns a specified FPA to a specified sector
- Deassign FPA from sector: deassigns a specified FPA from a specified sector
- Get assigned sector: returns an object reference to the sector to which a specified FPA is assigned
- Assign to FPA: assigns a specified FPA to another specified FPA
- Deassign from FPA: removes the assignment for a specified FPA from another specified FPA
- Get the FPAs that are assigned to this FPA: returns a list of the FPAs assigned to a specified FPA

In terms of our notation, we represent the FPA class as shown in Figure 22:

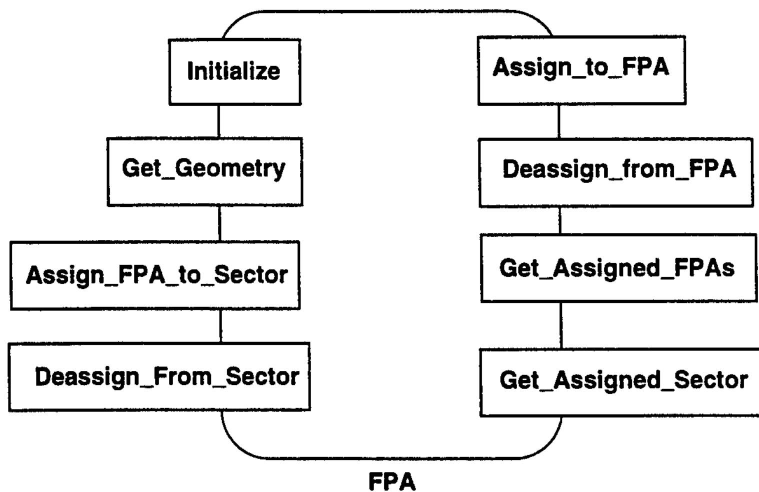


Figure 22: Initial Object Specification for a Fix Posting Area

4.5.4 Sectors

A sector is specified as a set of FPAs. A sector object will manage all information associated with a sector such as tracks, flight plans, and lists. When a sector object is created, it will create instances of inbound, hold, and conflict-alert lists. The sector object computes and maintains list data (such as inbound and hold lists) based on information provided from flight-plan management.

The state data associated with a sector include the following:

- geometry
- assigned console
- list of tracks (object references) in sector
- list of FPAs (object references) in the sector
- hold-list object reference
- inbound list object reference
- conflict-alert list object reference

The methods associated with the sector class include the following:

- Initialize: performs initialization for the sector
- Get assigned FPAs: returns a list of all FPAs assigned to the sector
- Assign sector to console: assigns a sector to a console and notifies the console object of the assignment
- Deassign sector from console: deassigns a sector from a console and notifies the console object of the deassignment
- Get assigned console: return the console object reference assigned to the sector
- Get sector geometry: returns the geometry information for this sector
- Assign FPA to sector: assigns an FPA to a sector object. The sector object shall notify the FPA object, the console object, and airspace-management objects.
- Deassign FPA from sector: deassigns an FPA from a sector object. The sector object shall notify the FPA object, the console object, and the airspace-management objects.
- Track notify: provides notification that either data are available for a new track or the data for an existing track have been updated
- Drop track: removes a track-object reference from the local track data
- Purge tracks: removes all track-object references from local track data
- Flight plan notify: provides notification that either data are available for a new flight plan or the data for an existing flight plan have been updated
- Delete flight plan: indicates that the flight plan has been deleted
- Purge flight plans: indicates that all flight plans have been removed from the system
- Conflict notify: provides notification of a new conflict or updated information about an existing conflict
- Conflict resolved: indicates that the conflict no longer exists
- Get list references: returns object references for all lists associated with this sector
- Get state data: returns all state data for the object

In terms of our notation, we represent the sector as shown in Figure 23.

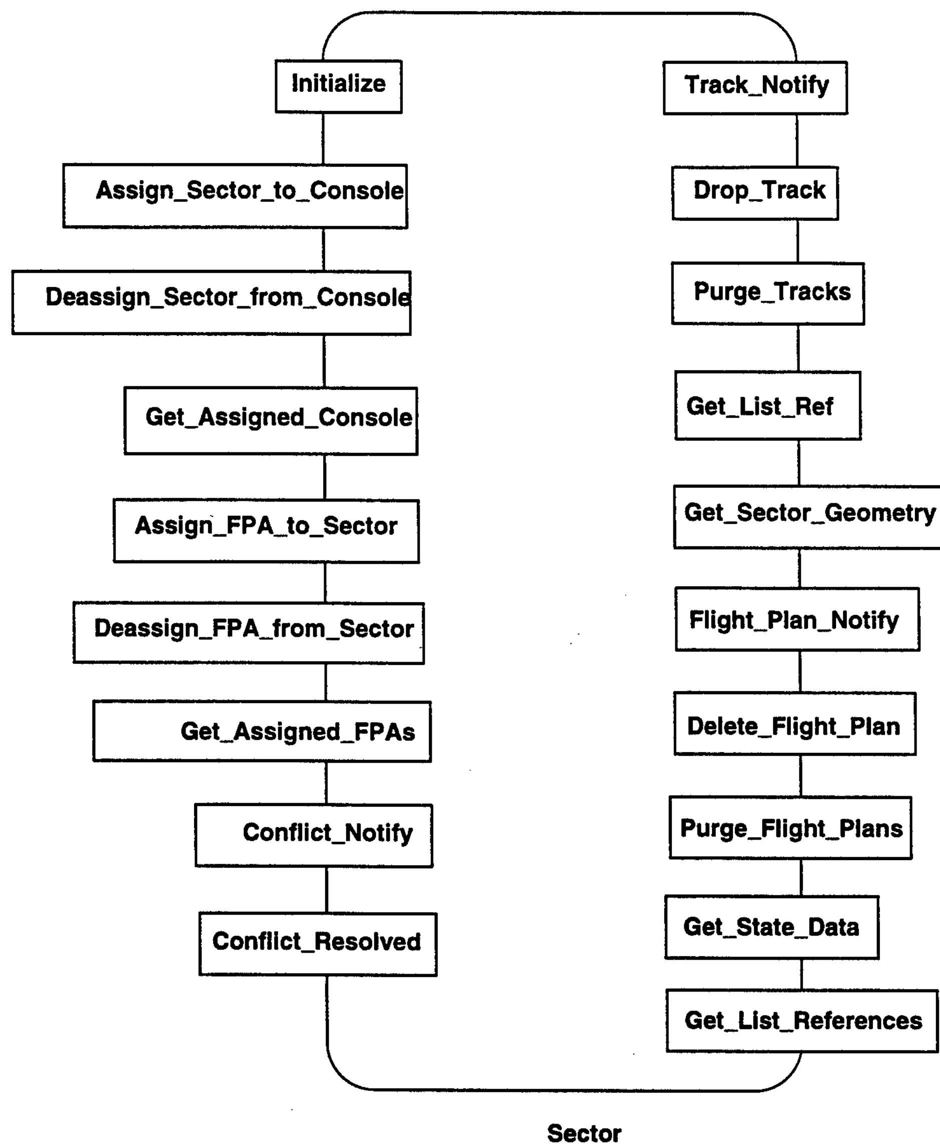


Figure 23: Initial Object Specification for a Sector

4.5.5 Consoles

The purpose of the console object is to manage information about the console display and sector assignment. As part of initialization, a console is expected to register with system management.

The state data associated with the console class includes the following:

- State: the current state of the console (active or inactive)
- Assigned sector reference: an object reference of the sector object that is assigned to the console
- Hold list reference: an object reference for the hold list associated with this console
- Inbound reference: an object reference for the inbound list associated with this console
- Conflict-alert list reference: an object reference for the conflict-alert list associated with this console
- Tracks: a database of object references for tracks associated with this console

The methods associated with the console include the following:

- Initialize: performs initialization for the console
- Assign sector: assigns a console to a specified sector
- Deassign sector reference: deassigns a sector object from the console
- Get assigned sector: returns an object reference of a specified sector
- Get console state: returns the current state for a specified console object
- Track notify: provides notification that either data are available for a new track or the data for an existing track have been updated
- Drop track: removes a specified track-object reference from the track-object reference database
- Purge tracks: removes all track object references from the track-object reference database
- Hold list updated: provides a notification of a data update in the hold-list object associated with the sector
- Inbound list updated: provides a notification of a data update in the inbound-list object associated with the sector
- Conflict-alert list updated: provides a notification of a data update in the conflict-alert list object associated with the sector
- Sector geometry changed: provides a notification that there has been a change in the geometry for a specified sector object reference

In terms of our notation, we represent the console class as shown in Figure 24.

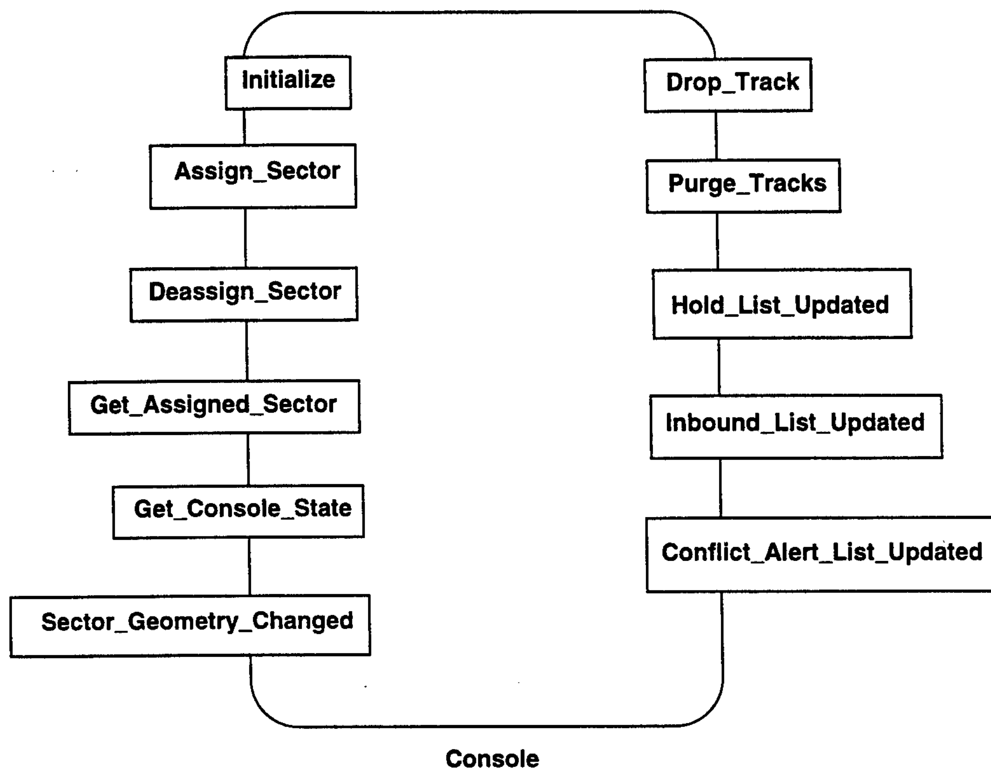


Figure 24: Initial Object Design for Console Class

4.5.6 Data Lists

A number of lists are associated with a sector. These include the inbound, hold, and conflict-alert lists which are discussed below.

4.5.6.1 Inbound List

The inbound list maintains a list of flight plans and associated parameters that provide information to the controller about flights that are expected to enter a particular sector.

The state data associated with the inbound list includes the flight-plan object reference.

The methods associated with the inbound list include the following:

- Initialize: performs initialization for the inbound list
- Add flight plan: adds a specified flight plan to an inbound list
- Delete flight plan: deletes a specified flight plan from an inbound list
- Clear list: clears all elements of an inbound list
- Modify list data: provides for changing the data associated with an inbound list

In terms of our notation, we represent the inbound list as shown in Figure 25:

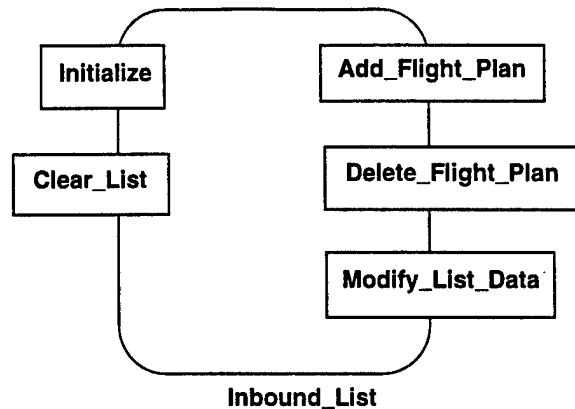


Figure 25: Initial Object Specification for an Inbound List

4.5.6.2 Hold List

A hold list contains information about flight plans that are currently in a hold state. The hold list is eligible for display at a particular console.

The state data associated with the hold list includes the following:

- flight ID
- hold fix
- hold time

The methods associated with the list include the following:

- Initialize: performs initialization for a hold list
- Add flight plan: adds a specified flight plan to a hold list
- Delete flight plan: deletes a specified flight plan from a hold list
- Clear list: removes all elements from a hold list
- Modify list data: provides for changing the hold-list data

In terms of our notation, we represent the hold list as shown in Figure 26.

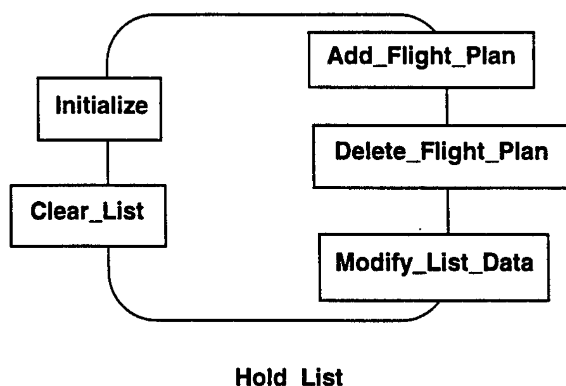


Figure 26: Initial Object Specification for a Hold List

4.5.6.3 Conflict-Alert List

A conflict-alert list maintains information about tracks that may be in a conflict situation.

The state data associated with the conflict-alert list includes the track pairs that are in conflict.

The methods associated with the conflict alert include the following:

- Initialize: performs initialization for the conflict-alert list object
- Add data: adds information to a conflict-alert list about track pairs that are in conflict
- Remove data: removes information from a conflict-alert list about track pairs that are in conflict
- Clear list: removes all data from the conflict-alert list

In terms of our notation, we represent the conflict-alert list class as shown in Figure 27.

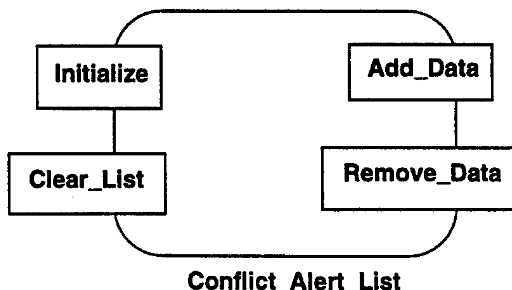


Figure 27: Initial Object Specification for a Conflict-Alert list

4.5.7 Track Management

The purpose of the track management object is to maintain information about each track object for the system. The airspace management object is informed of the existence of a new track, an update to an exiting track, or a track deletion.

The state data associated with track management includes a list of object references for all tracks in the system.

The only method exported by track management is that for initialization.

In terms of notation, the track management object appears as shown in Figure 28.

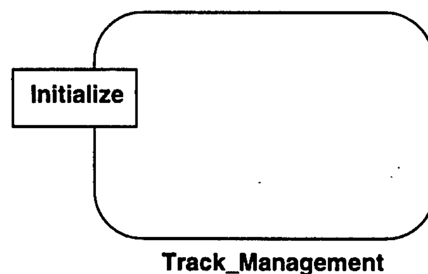


Figure 28: Initial Object Specification for Track Management

4.5.8 Track Data

A track is one of the fundamental data types used in an En Route system. Information about tracks are displayed for controllers and used to predict possible conflicts.

The state data associated with a track include the following:

- position (x and y)
- altitude
- speed
- heading
- computer ID
- flight-plan object reference

The above list is representative of track state data; details can be found in Section 3.3.4.1. The methods associated with the track object class include the following:

- Update track parameters: provides a way to update state data associated with a particular track
- Get track parameters: provides state data for a specified track
- Extrapolate track: extrapolates dynamic attributes of a track (such as position and velocity) ahead in time

In terms of our notation, we represent a track object as shown in Figure 29.

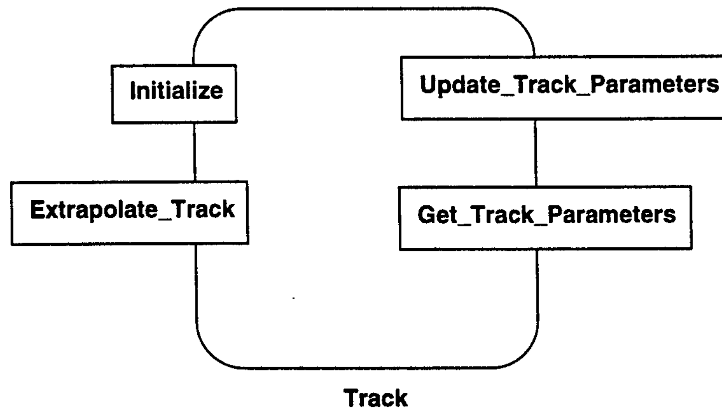


Figure 29: Initial Object Specification for a Track

4.5.9 Flight-Plan Management

The purpose of the flight-plan management object is to maintain information about each flight plan object for the system. The airspace management object is informed of changes to a flight plan.

The state data associated with flight-plan management includes a list of object references for all flight plans in the system.

The only method exported by flight-plan management is for initialization.

In terms of notation, the flight-plan management class would appear as shown in Figure 30.

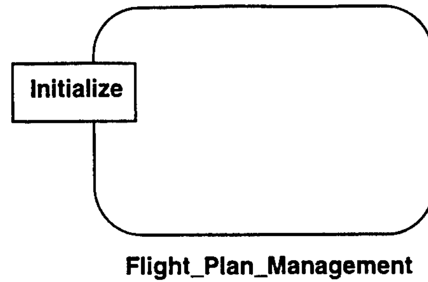


Figure 30: Initial Object Specification for Flight-Plan Management

4.5.10 Flight Plans

A flight plan is one of the basic data elements fundamental to the operation of an En Route center.

The state data associated with a flight plan includes the following:

- aircraft ID
- aircraft data
- beacon code
- speed
- coordination fix
- coordination time
- assigned altitude
- requested altitude
- route

The methods associated with the track class include:

- Initialize: performs initialization for a flight plan object
- Update FP data: provides for updating information about a specified flight plan
- Get FP data: provides for data for a specified flight plan

In terms of our notation, we represent the flight-plan class as shown in Figure 31.

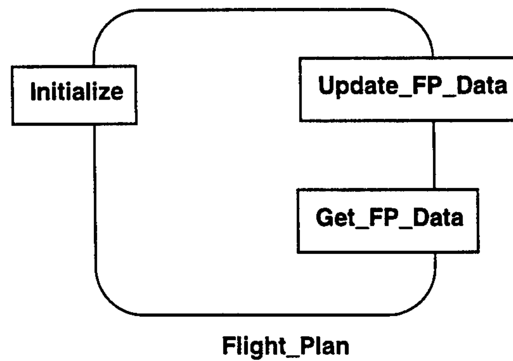


Figure 31: Initial Object Design for Flight-Plan Object

4.5.11 Summary of Initial Design

4.5.11.1 Objects and Their Interaction

A simple summary of the initial design is obtained by considering object (classes) and their interactions (via method invocations). This information is presented in Figure 32. For purposes of brevity, we do not show the methods on this diagram.

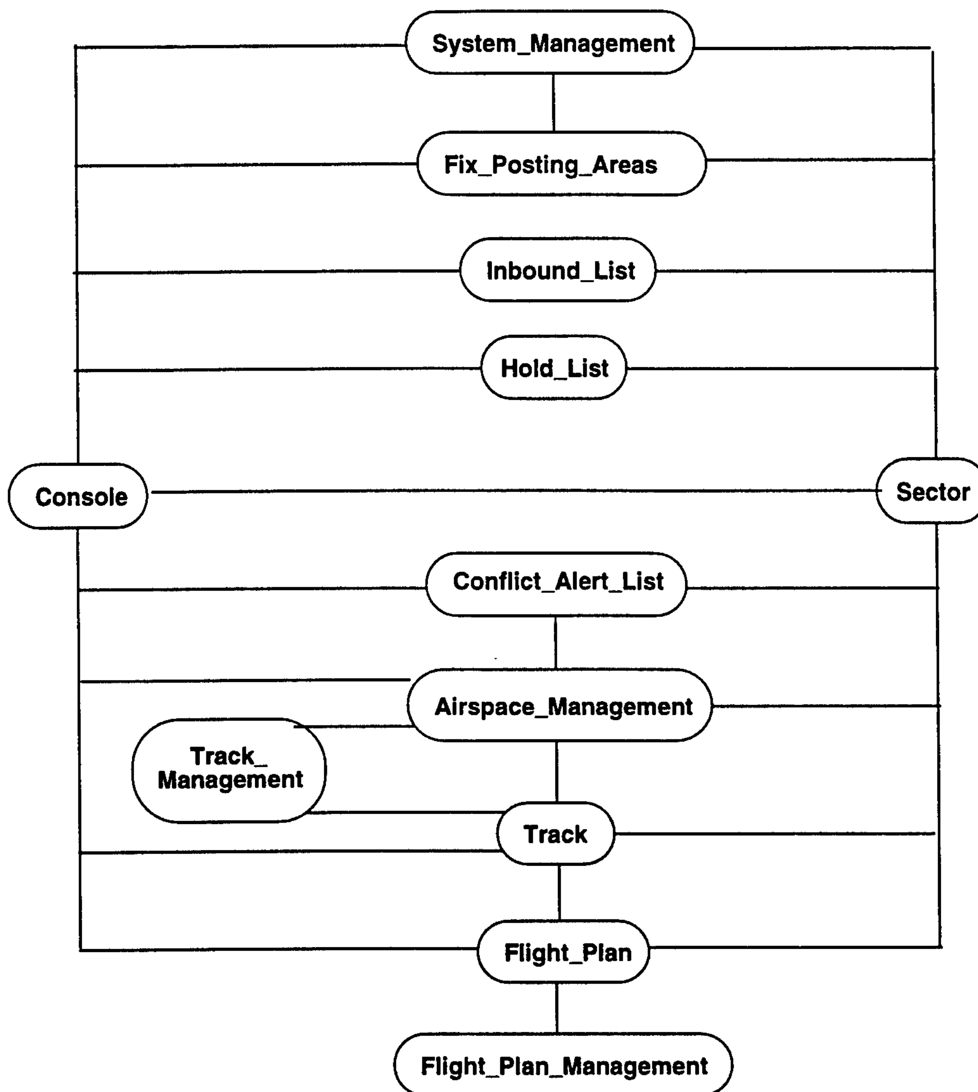


Figure 32: Initial Design Objects and Interaction

Recall that we have made the assumption of a steady state when we developed the above design. Among other things, the design does not address initialization requirements. In reality, we would expect interaction between system management and all other objects. One reason for this would be related to initialization, while another would be to maintain and display state data for system-managed objects. For example, it may be desired to display the number of conflict pairs for each sector as part of system management.

An interesting question is to consider the number of objects that would be present in an En Route system. This is presented in the Table 3.

Object	Instances
System Management	2
Airspace Management	1
Fix Posting Areas	655
Sectors	100
Consoles	100
Inbound List	100
Hold List	100
Conflict-Alert List	100
Track Management	1
Tracks	700
Flight-Plan Management	1
Active Flight Plans	2500

Table 3: Estimated Number of Objects in Initial Design

Table 3 indicates a total of about 4,000 objects in the system. Note that the above estimate is largely based on system capacity requirements (see Table 1).

4.5.11.2 Sample Data Flows

In this section, we will present some typical data flow diagrams for the initial CORBA design. We do this for several reasons, including to

- help the reader understand the functionality of the objects and their interaction
- gain insight into issues that may play a role in the refinement of the initial design

Data flows will be presented for the following cases:

- track update
- flight-plan update
- sector combination
- console-object failure

4.5.11.2.1 Track Data

The first example of a data flow consideration will be that for the distribution of track data. We chose this for a number of reasons. First, the ability to display track data in a timely manner is a fundamental requirement of the En Route system. Second, when we consider the problem of sector combination, it requires a look at the redistribution of track data. Hence, an understanding of how track data are distributed is important for other functions performed in the En Route center.

A diagram representing the data flow for an update of a track is shown in Figure 33.

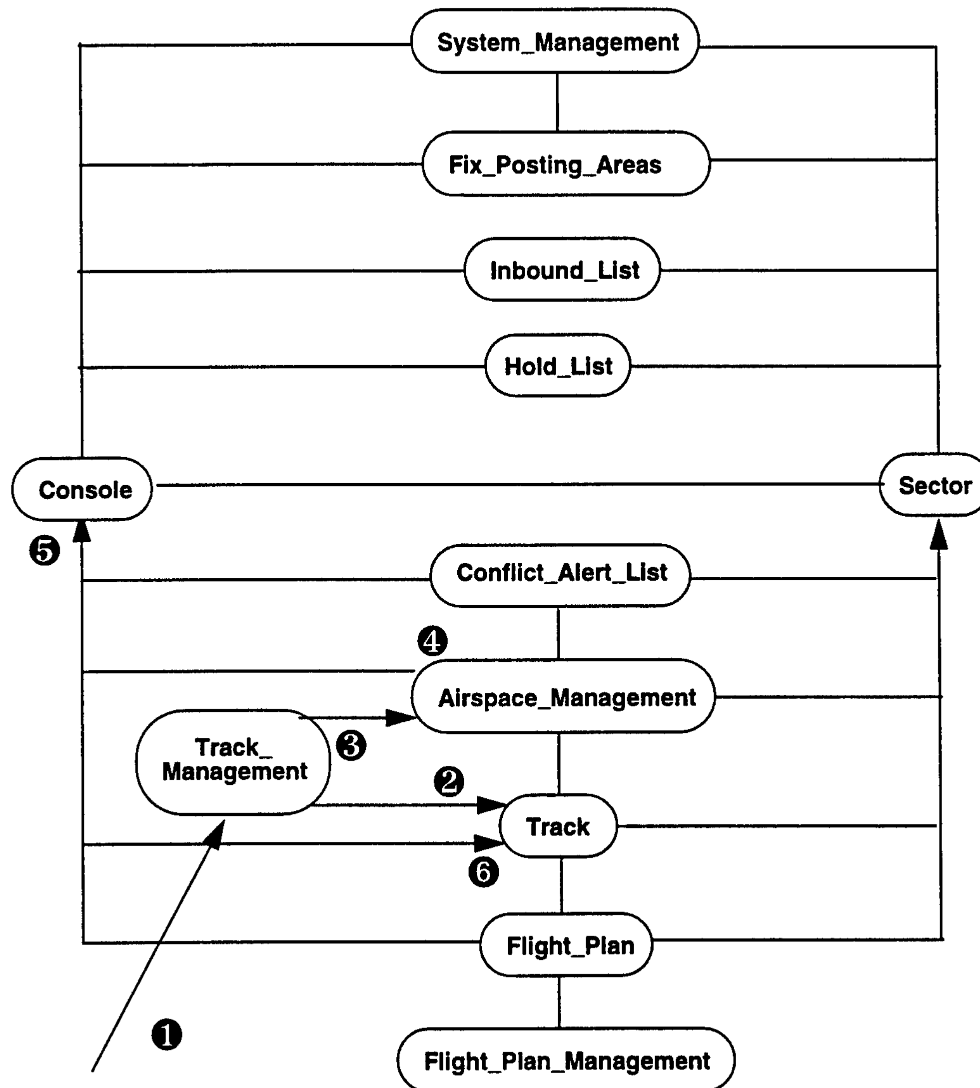


Figure 33: Initial Design for Track-Update Data Flow

The sequence of events for the track update is described below (items in the list below correspond to those in Figure 33):

1. Get track object reference: Data are provided from the radar, and we assume that there is a computer ID (CID) for this track. The CID may be used by track management to determine an object reference for the particular track that is being updated. The mapping of CID onto track is maintained.
2. Update track with data: Track management invokes a method on the object reference for the specified track. The particular method is that which updates the state data for the track, such as position and velocity information.
3. Get sector object reference for track: Track management invokes a method on airspace management to obtain an object reference for the sector(s) where the track is being displayed.
4. Get consoles for track display: Airspace management must determine the sectors where this track information must be displayed.
5. Distribute object reference for track: Airspace management invokes a method on the console object to provide an object reference for this track.
6. Get and display track data: The console where the track will be displayed uses the object reference to invoke a method on the track object. This is done to obtain information about the track parameters. Once the information is returned, the new track data are displayed.

It is an interesting exercise to develop a very rough estimate of the time required to update a track on a display console. Items 2 through 6 in the above list will each require a remote procedure call, meaning five remote procedure calls in total. If each remote procedure call takes 4 milliseconds, it will require 20 milliseconds to update one track.

The system requirement is for 640 tracks which are updated during a 6-second radar scan. This amounts to about 110 tracks per second. If each track requires 20 milliseconds, it means a rate of 50 per second. Hence, a very rough estimate indicates that tracks can be updated at only half the required rate.

Note the sensitivity to the assumed value of the time for the remote procedure call. If the time is 50 percent too large, it is possible to update all tracks in the specified interval. On the other hand, if the time for a remote procedure call is 50 percent too small, the requirement rate is off by a factor of 4.

The preceding has been a very rough estimate of the overall track-update cycle. It was based only on an assumed value for a remote procedure call. It did not take into account any other concurrent operations that could block the track update. Such considerations would require a more detailed analysis.

4.5.11.2.2 Flight-Plan Data

A second important category of data is that associated with a flight plan. We will consider the case where an amendment message arrives at the En Route center for an existing flight plan. The relevant data flow diagram is presented in Figure 34 below.

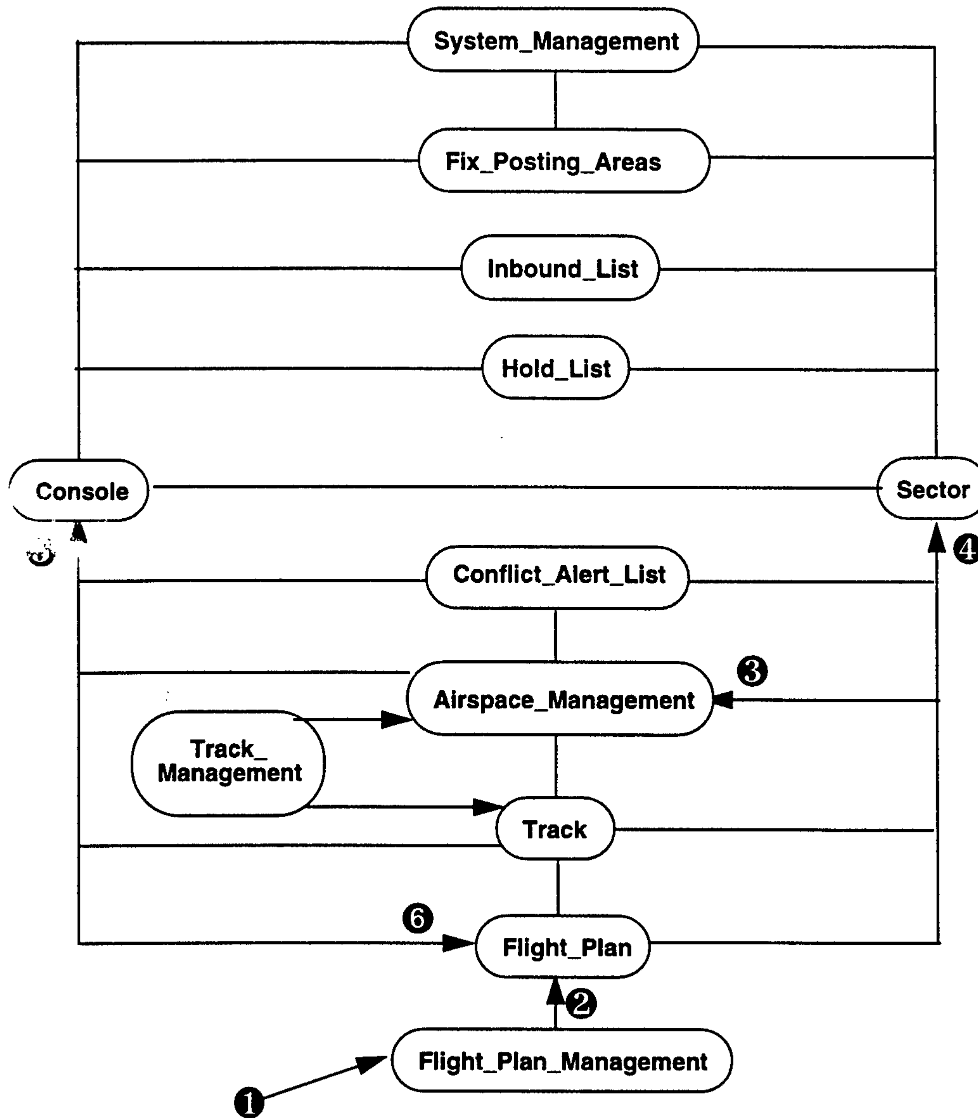


Figure 34: Initial Design for Flight-Plan Update Data Flow

The sequence of events for the update to a flight plan is described below:

1. Get flight-plan object reference: When the amendment message is received, we assume it is sent to flight-plan management for processing. The object reference for the specified flight plan is determined. The mapping of flight plan ID to flight object is maintained.
2. Update flight-plan data: The obtained object reference is used to update the flight- plan data for the object.
3. Get sectors: A given flight plan can be resident in one or more sectors.
4. Get consoles: Given the sectors on which the flight plan data is present, it is necessary to determine the display consoles where the sector resides.
5. Distribute object reference to consoles: Each console that has knowledge of the flight plan must have an object reference for the specified flight plan object.
6. Get and update flight-plan data: Each console that is holding the flight plan uses the object reference to invoke a method on the flight-plan object. This results in the distribution of new data to the console where it can be displayed, and so forth.

4.5.11.2.3 Sector Combination

The initial motivation for this work was to consider the question of sector combination. We will now describe the data flow and sequence of operations that occur when there is a choice made to combine two sectors. This situation is presented in the Figure 35.

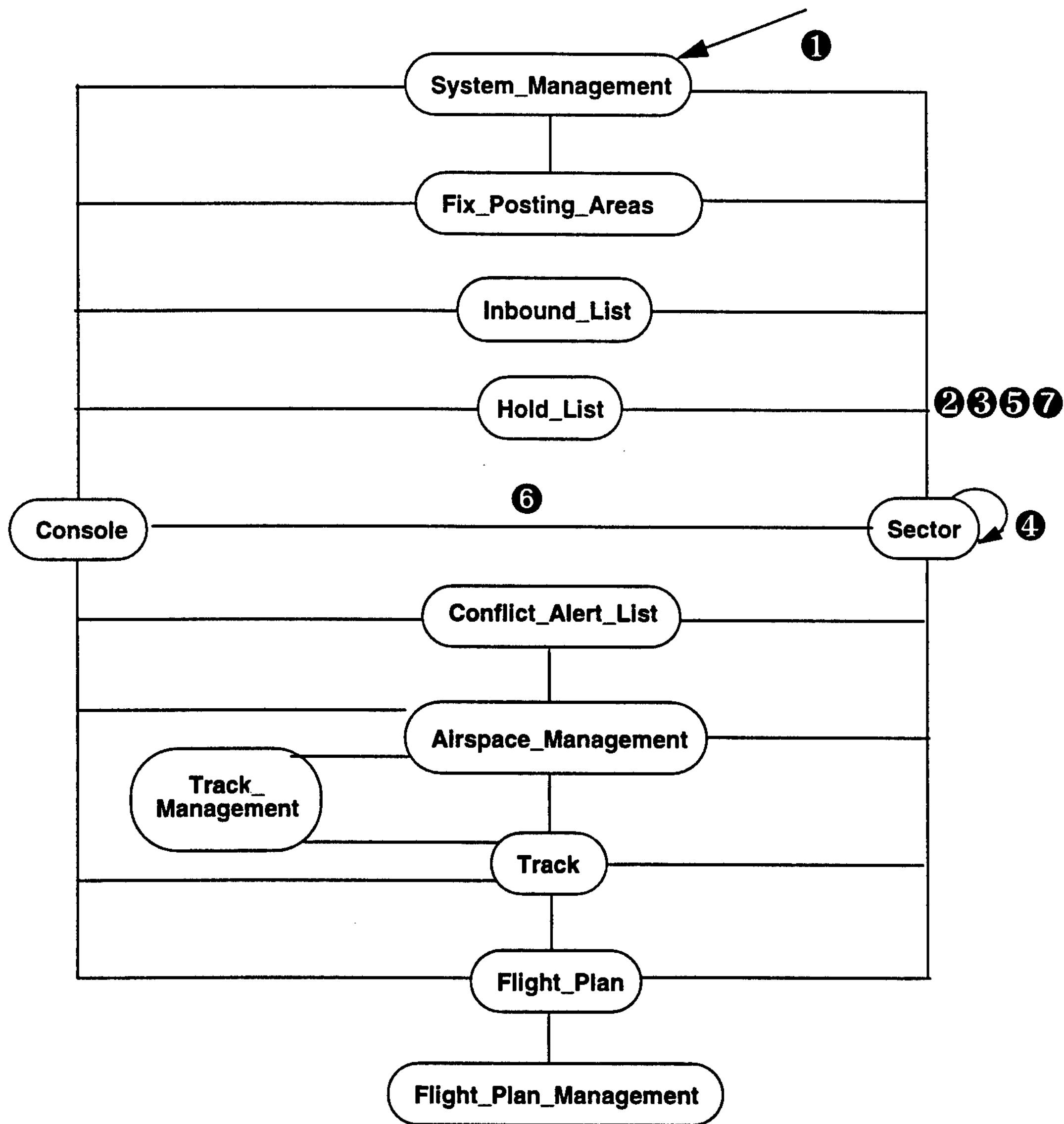


Figure 35: Initial Data Flow for Sector Combination

The sequence of events for sector combination is described below:

1. Operator request: A system-management operator invokes the request to combine the specified sectors. For example, consider combining sector SLwith sector SR.
2. Create new sector object: System management creates a new sector (say S) that will be responsible for controlling the areas previously controlled by sectors SL and SR. This will involve the creation of other new objects such as hold, inbound, and conflict-alert lists.
3. Distribute object references for existing sectors: The system-management object invokes a method on the newly created sector and provides it with the object references for the sectors that are to be combined.
4. Get sector data: The newly created sector invokes methods on sectors SL and SR to retrieve their FPAs, inbound lists, conflict-alert lists, track data, and flight-plan data. After the sector data have been retrieved, sectors SL and SR will not accept new data. On completion, the new sector, S, informs the system manager that it has captured the data from sectors SL and SR.
5. Activate new sector: Either the system manager activates the new sector and calls, or data destined for sectors SL or SR are directed to the combined sector, S.
6. Assign new sector to console: The new sector is assigned to the appropriate controlling console.
7. Delete sector objects: The last step in this process is to delete the sector objects associated with sectors SL and SR that were combined to form the new sector. As part of deleting the sector objects, we assume that the associated objects, such as the hold-list object, inbound-list object, and so forth, are also deleted.

4.5.11.2.4 Console Object Failure

Fault tolerance is an important consideration in the En Route domain. Hence for this final example, we will consider the case where there is a failure of a console object that is in an active state. Since this initial design is for a virtual machine, we are restricted to discussing the failure of an object, as opposed to a physical device. Thus, the failure of a console object is taken to illustrate just one of several possible failures that could occur. It will provide us with an overall understanding of what happens in the case of a failure and the processing that needs to be performed for detection and recovery. For this case, we assume that spare display consoles that have software loaded exist, but are not controlling a particular sector. The relevant data flow diagram is shown in Figure 36.

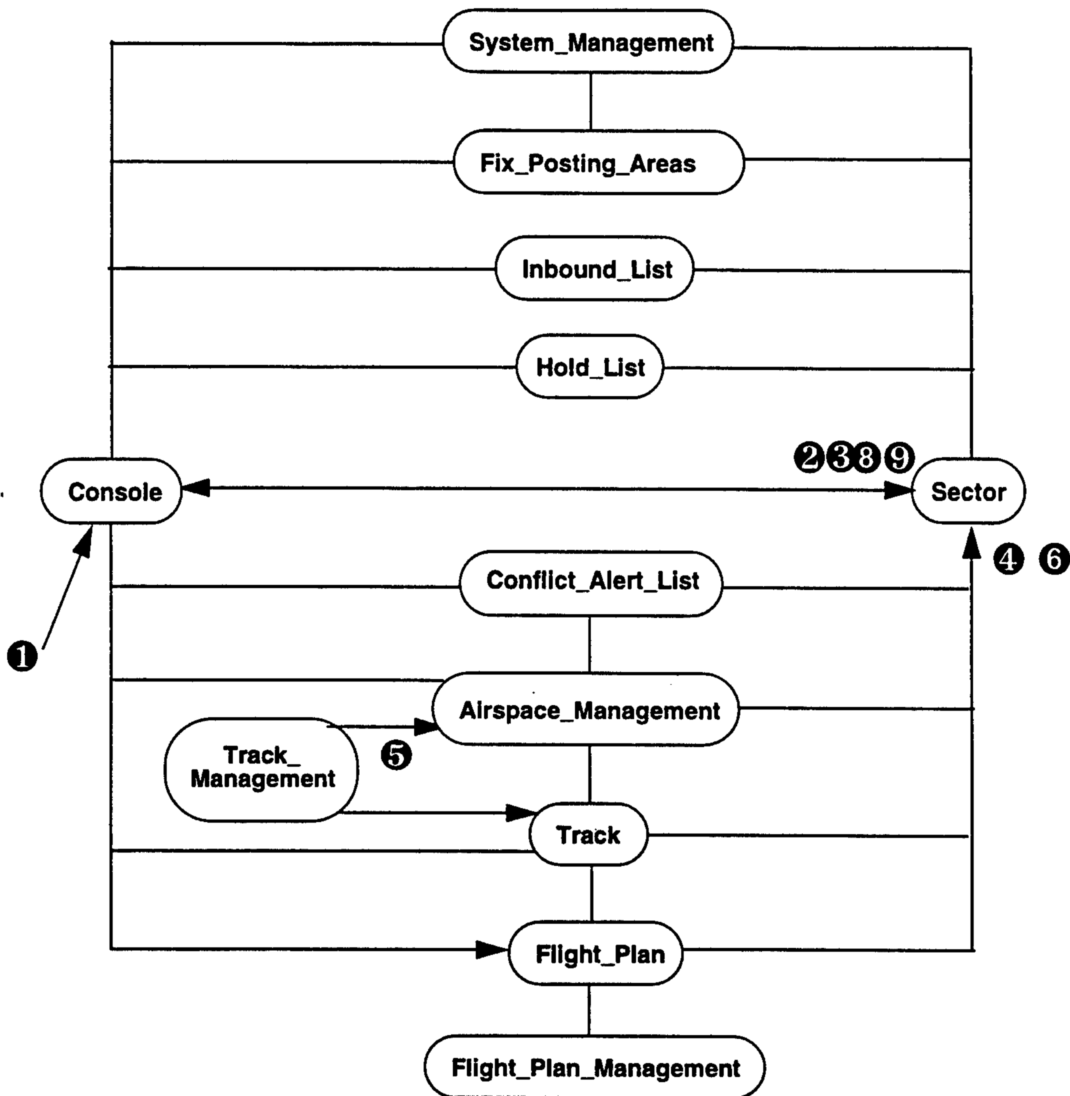


Figure 36: Initial Design for Data Flow of Console-Object Failure

The sequence of events associated with handling a display-console failure is described below:

1. Console fails: An active display console undergoes a hard failure such that it is incapable of sending and receiving any data.

2. Failure detected: Some object attempts to invoke a method on an object resident in the failed console. For example, assume that the assigned sector object attempts to notify the Console Object that the inbound list has been updated. This will require the invocation of a method on the failed console object. Note that the object reference is valid; it is the device referenced by the object reference that has failed. The method will fail, and a CORBA exception will be raised.
3. Deassign sector: The deassign sector method is invoked on the airspace-management object by the system-management object.
4. Deassign sector: Airspace management invokes the deassign-sector method on the sector object that is assigned to the failed console.
5. Assign sector: The system-management object invokes the assign-sector method on the airspace-management object.
6. Assign sector: Airspace management invokes the assign-sector method on the sector object to be assigned to the console.
7. Assign sector: The sector object invokes the assign-sector method on the console object.
8. Get sector geometry: The console object invokes the get-sector geometry method on the assigned sector object.
9. Get list object references: The console object invokes the get-lists-object-references method on the assigned sector object.

4.6 Refinement of Initial Design

In this section, we will refine the initial design based on considerations of performance and fault tolerance. For each object class presented in Section 4.5, we will reassess the initial design. For example, we are interested in questions such as the following:

- Are there performance considerations that would warrant an initially selected object (class) not to be treated as a CORBA object?
- Are there fault-tolerant considerations that would warrant an initially selected object (class) to not be treated as a CORBA object?

4.6.1 System Management

The purpose of the system-management object is to encapsulate state data for the system. In the refinement of the design, we believe that this choice continues to be appropriate. It maintains a list of all console object references and sectors with assigned FPAs.

The state data associated with system management includes the following:

- console object references: a database of system console object references
- sectors with assigned FPAs

The methods associated with the system management include the following:

- Initialize: performs initialization for system management
- Console update: indicates that a new console is now online, or an existing console has changed state
- Sector update: indicates that a sector has been created, deleted, or changed
- FPA update: indicates that an FPA has been modified

In terms of our notation, we represent the system management as shown in Figure 37.

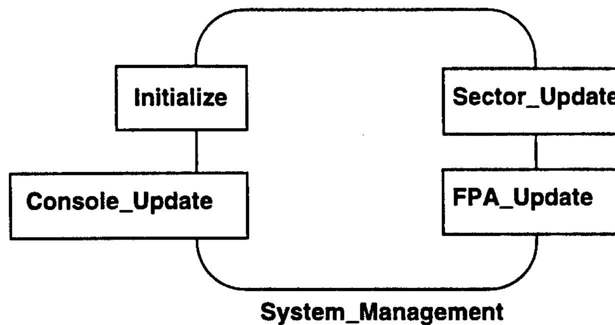


Figure 37: Refined Object Specification for System Management

4.6.2 Airspace Management

The purpose of airspace management is to manage the creation, modification, and deletion of sectors. This function also distributes data to console objects based on the sector and console geometry. The distribution of track data to this object is performed by the track management object. As part of system initialization, all display and management consoles must register with airspace management.

The state data associated with the airspace manager object includes the following:

- database of all sectors in the system with the assigned console and FPAs
- database of all display consoles with the console's geometry and assigned sector
- database of references for all registered management console's

The methods associated with the airspace manager class include the following:

- Initialize: performs initialization for the airspace-manager object
- Register display console: provides for a console, specified by an object reference, to register with airspace management
- Register management console: provides for a system management console, specified by an object reference, to register with airspace management
- Get registered display consoles: returns a list of registered consoles along with state information
- Create sector: creates a sector from a specified set of FPAs
- Delete sector: deletes a specified sector and deassigns all FPAs associated with that sector
- Get all sectors: returns a list of all sectors in the system with the assigned FPAs and console object references
- Get all FPAs: returns a list of all FPAs in the system with the FPA geometry and assigned sector
- Combine sector request: requests two sectors (or airspace entities, such as FPAs) to be combined
- Combine sectors: combines two (or more) sectors into one sector
- Split sectors: removes a previously established sector combination (i.e., to split a sector into its constituent parts)
- Assign sector: assigns a sector to a console
- Deassign sector: deassigns a sector to a display console
- Assign FPA: assigns an FPA to a sector
- Deassign FPA: removes an FPA from a sector
- Track update: creates new track or update of an existing track
- Drop track: track that must be dropped
- Purge tracks: all tracks dropped
- Flight plan update: a new flight plan or an update for an existing flight plan
- Delete flight plan: indicates a flight plan that needs to be deleted
- Purge all flight plans: indicates that all flight plans are no longer valid
- Conflict update: indicates a new conflict or an update of an existing conflict
- Conflict resolved: indicates that the conflict no longer exists
- Console geometry update: provides notification of a change in the geometry of a console

In terms of our notation, we represent the airspace-manager object as shown in Figure 38.

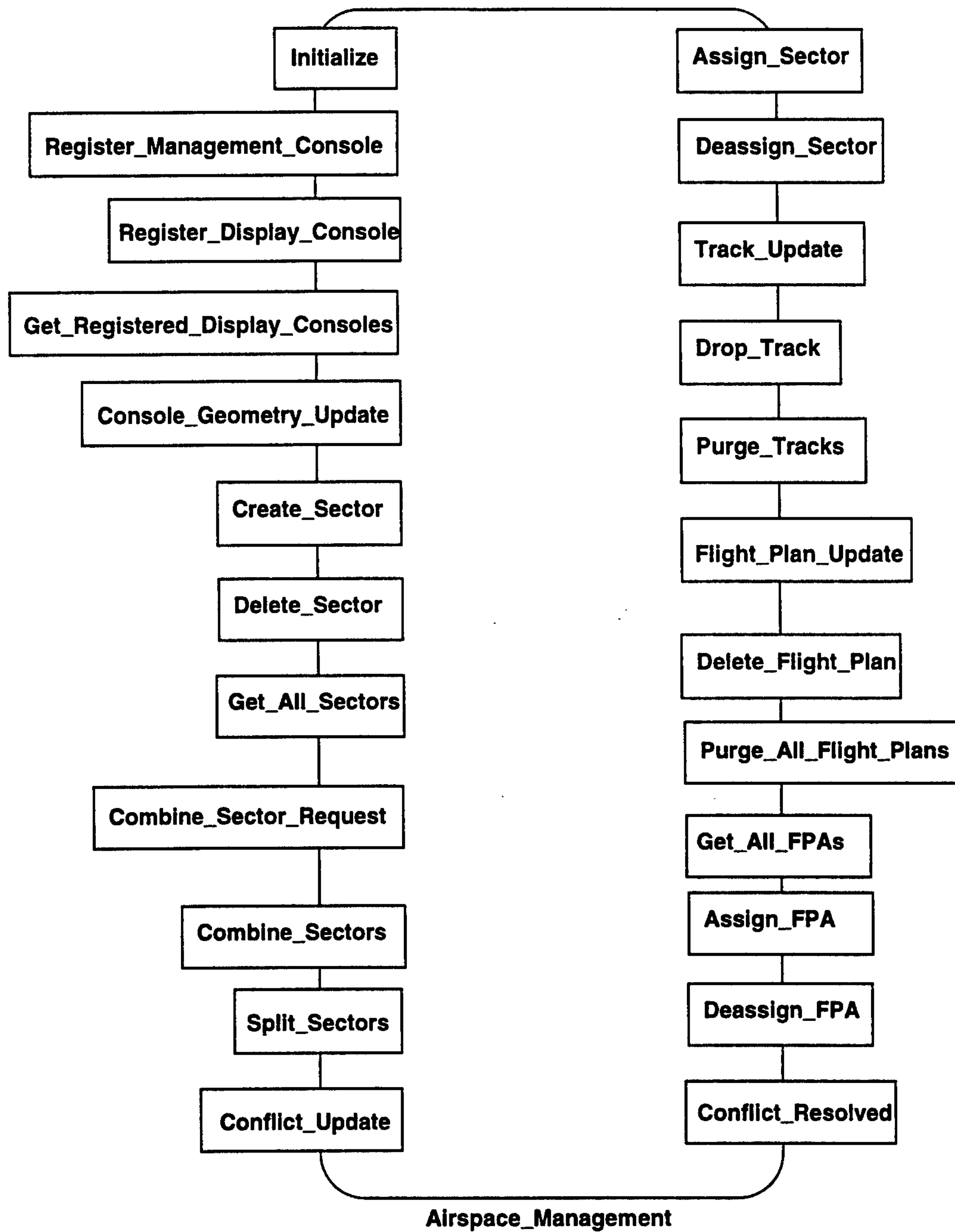


Figure 38: Refined Object Specification for Airspace Management

4.6.3 FPAs

In the initial design, each FPA was treated as a separate object. In the refinement, we believe that this is not the correct choice for the following reasons:

- FPAs are static (i.e., their state data are fixed as part of adaptation data processing for system initialization).
- FPAs are small in functionality. (For example, there are a small number of methods associated with an FPA in the initial design; see Figure 22.)
- Often, it is necessary to deal with an aggregate of FPAs rather than an individual FPA (e.g., assigning one sector to another can be represented as assigning all FPAs in one sector to another sector).

Based on the above considerations, we make the following decision:

Design Consideration C-4
In the refinement of the design, fix posting areas will not be represented as CORBA objects.

Note that a major influence in the above decision was the static nature of the FPA. For example, if the FPA were a dynamic object, a node that forms one of the vertices of the FPA could be moved; then the above decision would be worth revisiting.

In the refinement of the design, we shall simply consider FPAs as a set of state data managed by airspace management.

4.6.4 Sectors

In the initial design, a sector was implemented as an object. In the refinement of the design, we believe it is not necessary to retain this choice for reasons similar to those discussed in Section 4.6.3. Hence, we have the following design considerations:

Design Consideration C-5
In the refinement of the design, sectors will not be represented as CORBA objects.

In the refined design, a sector will be treated as having state data and managed by airspace management.

4.6.5 Consoles

The purpose of the console object is to manage information about the console display and sector assignment. As part of initialization, a console is expected to register with airspace management.

The state data associated with the console class includes the following:

- state: the current state of the console (active or inactive)
- assigned sector geometry
- console geometry
- hold list
- inbound list
- conflict-alert list
- tracks database
- flight-plan database

The methods associated with the console include the following:

- Initialize: performs initialization for the console
- Activate console: sets console to an active state
- Deactivate console: sets console to an inactive state
- Set sector geometry: sets the sector geometry of the console
- Track update: creates new track data or a data update for an existing track
- Drop track: removes the track from the database
- Purge tracks: removes all tracks from the track database
- Conflict-alert list update: updates conflict data
- Flight plan update: creates new flight-plan data or a data update for an existing flight plan
- Flight plan drop: removes a flight plan from the database
- Purge all flight plans: indicates that all flight plans are no longer valid
- Get console state: returns the current state of a specified console

In terms of our notation, we represent the console class as shown in Figure 39.

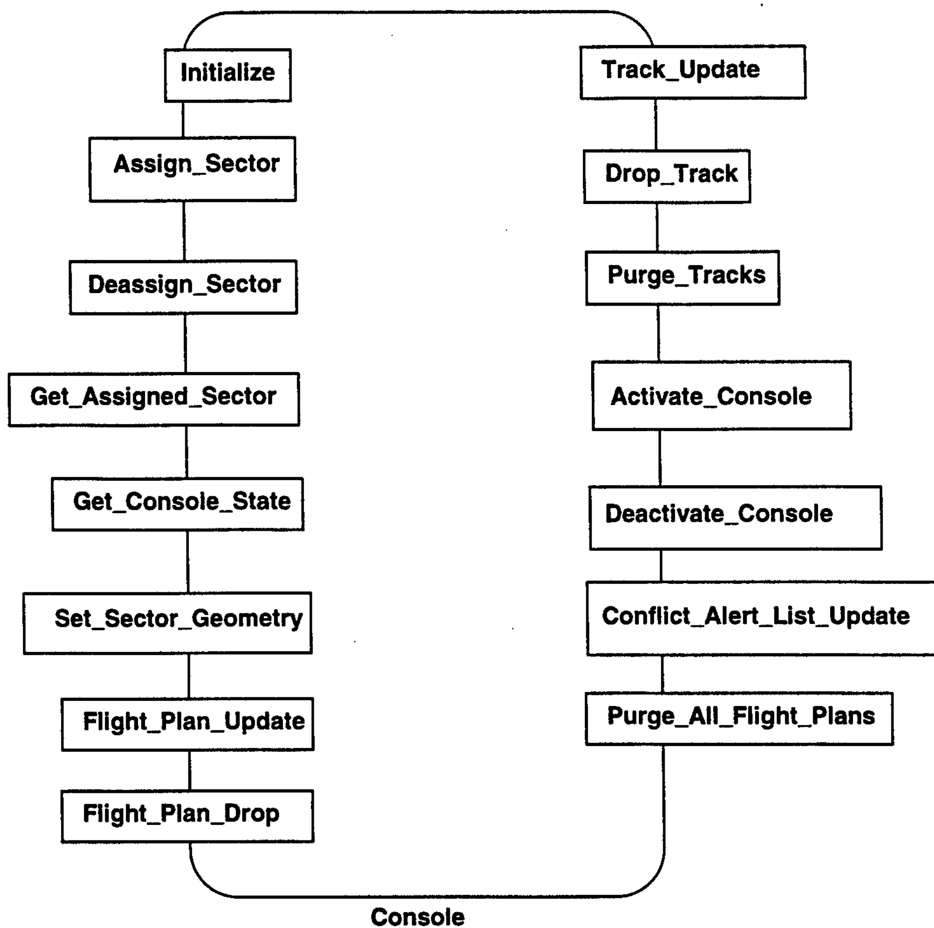


Figure 39: Refined Object Design for Console Class

4.6.6 Data Lists

A number of lists are used to display information on a particular console. Examples of this include hold, inbound, and conflict-alert lists. It is necessary to decide if these lists will be treated as objects or as an attribute of an object.

Each of the lists noted is directly associated with only one sector. In addition, only one instance of a list exists for each sector. These considerations lead us to the following design consideration:

Design Consideration C-6
All sector-specific lists, such as hold, inbound, and conflict-alert lists, will be treated as attributes of the sector with which they are associated.

The preceding result indicates that lists, such as the inbound list, will not be CORBA objects. However, it is necessary to maintain the interface for each list. That is, the inbound-list object had a method that allowed a flight plan to be added to the list.

4.6.7 Track Management

The purpose of track management is to encapsulate information about tracks. This is achieved in the initial design and will be retained in the refinement of the design. There is only one object for the track-management function. A later possibility of design modification would be to have separate track managers for different types of tracks. For example, it would be possible to have a track-management object for managing *simulated* tracks. If that were the case, it would allow for data encapsulation and allow a natural separation of real and simulated tracks. Further consideration of simulated tracks is beyond the scope of this report.

Track management maintains the following state data:

- objects registered to receive track data and the type of information requested
- conflict list
- a track database with the following state data for each track:
  - position (x and y)
  - altitude
  - speed
  - heading
  - computer ID
  - flight-plan cross reference

The above list is representative of track state data (for details, see Section 3.3.4.1). The methods associated with the track-management class include the following:

- Initialize: performs initialization for track management
- Track data register: instructs track management to provide track data to an object
- Track data de-register: instructs track management to stop providing track data to an object
- Correlate track: associates a flight plan with a track

- De-correlate track: disassociates a flight plan with a track
- Track update: provides track data from a sensor
- Get tracks for region: provides all track data for a specified region; used for sector combination and other airspace modification requests
- Get conflicts in region: returns a list of conflicts within the specified region

In terms of our notation, the track-management object appears as shown in Figure 40.

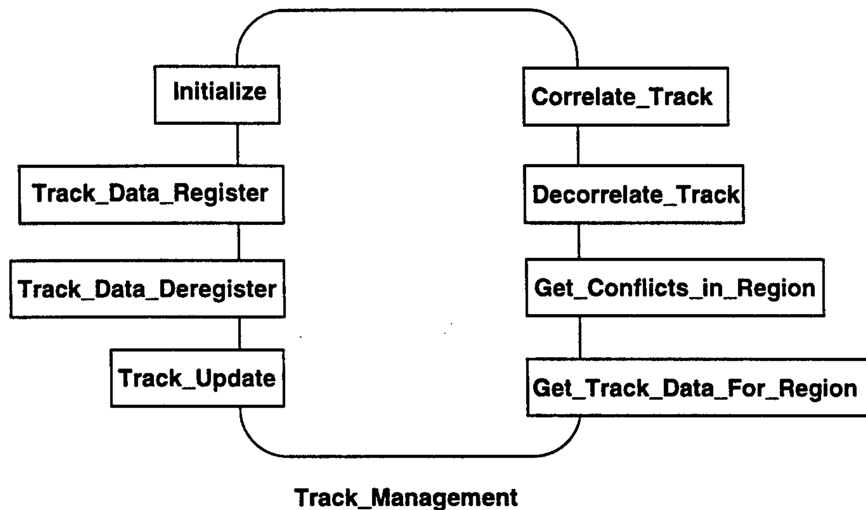


Figure 40: Refined Object Specification for Track Management

4.6.8 Track Data

4.6.8.1 Object Considerations

In the initial design, each track was instantiated as an object. In the refinement of the initial design, it is worthy to reconsider this choice. We believe that this is *not* an appropriate design choice for the following reasons:

- Managing over 500 individual objects may have adverse performance consequences.
- There are functions, such as determining track conflict, which require comparing one track against all others. This could mean *n* method invocations of the *n* track objects in order to determine the existence of a possible conflict. For a value of *n* equal to 640 tracks and a 2 millisecond method invocation time, this would mean 1.28 seconds simply to get the data, even before any conflict-alert processing was initiated. Furthermore, during the time required for all the communication of track data (methods to get the state for each track, for example), the state would be changing.

In view of the above consideration, we make the following design consideration:

Design Consideration C-7 Tracks will not be implemented as CORBA objects.

Instead, tracks will be managed in some other manner, such as a centralized data structure that would permit database operations to get and set attributes of a track.

4.6.8.2 Local or Distributed Management

A basic question that deals with track data is the ability to maintain state. Another way to consider this question is whether the data are managed in a centralized manner or distributed among a number of components. There are two possible choices:

- Have a central management capability for track objects.
- Have each track object managed by the console(s) where the track is displayed.

One advantage to a centralized management approach is overall maintenance of state data. A related advantage is that when there is a need to perform an operation over multiple pairs of tracks (such as conflict-alert processing), it is more efficient if those data are centrally managed.

On the other hand, it is possible for track-data to be distributed. One possibility is that when new data are received for a track, those data are transferred to the appropriate display console. Thus, the distributed management would be on a per-console basis. However if a console fails, it would not be possible to recover the current state of those tracks that were maintained by the failed console. This would further mean, for example, that until the data could be reconstituted (by new radar reports most probably), there would not be data available to display or to use with track conflict-alert processing.

The preceding discussion leads us to make the following design consideration:

Design Consideration C-8 Track data will be managed in a centralized manner.

From the perspective of an object-oriented design, the result of this design consideration is that track objects will be physically collocated. We assume further that they are resident in a device that is fault tolerant and capable of restoring a state. Note that we could have made a similar choice and required, for example, that all display consoles be fault tolerant and capable of maintaining a state. However, for practical reasons, we felt that it would be easier to have just one track-management capability be fault tolerant, rather than perhaps 100 fault-tolerant consoles.

4.6.8.3 Distribution of Track Data

A fundamental characteristic of an En Route center is the ability to distribute track data to one, or more, display consoles. There are two ways in which track data are distributed in existing systems:

1. Distribute track data to all consoles, and then each console would filter the data as necessary.
2. Send the track data to a console on an as-needed basis. That is, if a console is expected to hold information on some set of tracks, only those tracks would be sent to that particular console.

Our first design choice for distribution of track data is based on the following design consideration:

> **Design Consideration C-9**
> Track data will be distributed only to those consoles that require information about a particular track.

The main reason for the above choice is to minimize the number of method invocations for distributing track data. There is clearly a performance penalty when providing all tracks to all consoles via unicast method invocations.[1] It is appropriate to note that the option of providing all tracks to any console is currently implemented using a broadcast capability.[2] However, CORBA does not provide such a mechanism. This further strengthens the rationale for distributing tracks only to those consoles that require information about a specific track. Typically, a given track could be displayed on two consoles, but this could be handled by invoking a method for the same track on each console. In other words, there is not a scalability question for distributing track data with respect to the consoles that may need track information.

A number of options are available to achieve the distribution of track data, including the use of methods or event services.

---

1. However, buffering track data may decrease the number of method invocations, but then one may either increase the end-to-end time to get data to a console or place timing constraints on the processing along the "string" for track-data distribution (e.g., requiring airspace management to have less time to process track messages).

2. It may be possible to achieve a functional equivalent of broadcast in a CORBA design, although we expect that the performance penalty could be quite high (e.g., replicated invocation of methods for all consoles). In such a case, scalability would be a serious issue.

We will not consider the use of services that are under development in the OMG, such as asynchronous messaging services.

4.6.8.3.1 Use of Methods

Given that tracks will be distributed only to those consoles that require information about a specific track, there are a number of ways to achieve this capability in CORBA. For the use of methods, there are the following choices:

- A console invokes a method on the track-data management object, requesting information about tracks.
- The track-data management object invokes a method on a console object and transfers information about tracks.

Note that the above options are the converse of each other. The first case is summarized in Figure 41.

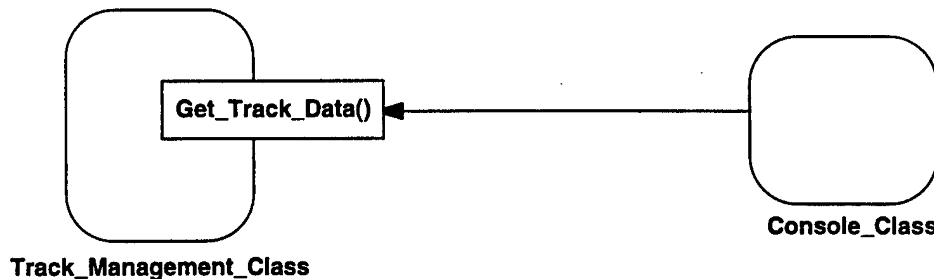


Figure 41: A Console Requesting Track Information

The case of the second design choice, where airspace management distributes track data (since airspace management has knowledge of the console geometry), would be represented as shown in Figure 42.

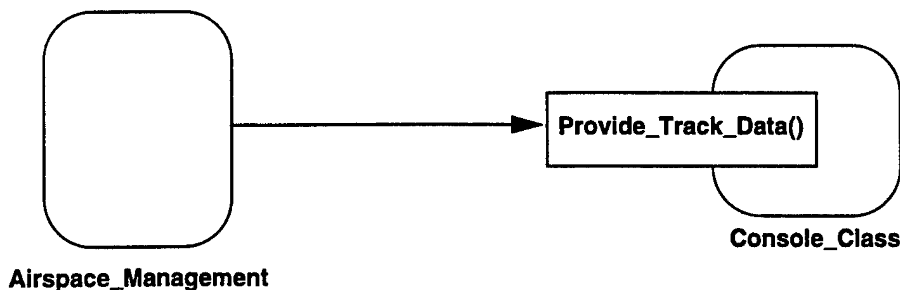


Figure 42: A Track-Data Management Object Transferring Track Data to a Console

There is an important distinction between the above two cases. In the case where the console initiates a method, this is tantamount to polling for track data by a *console*. It is well known that there are performance consequences associated with the use of polling. In view of this, we make the following design consideration:

> **Design Consideration C-10**
> If the distribution of track data will be accomplished through the use of a method invocation, the receiver of the data should be invoked by the distributor of the data, and not the converse.

Note that the number of possible method invocations would be equal to the number of receivers (in this case, about 50 consoles) multiplied by the number of methods (about 5, one for *Drop Track*, *Update Track*, etc.). Hence, a somewhat more realistic picture that emerges is like the one shown in Figure 43.

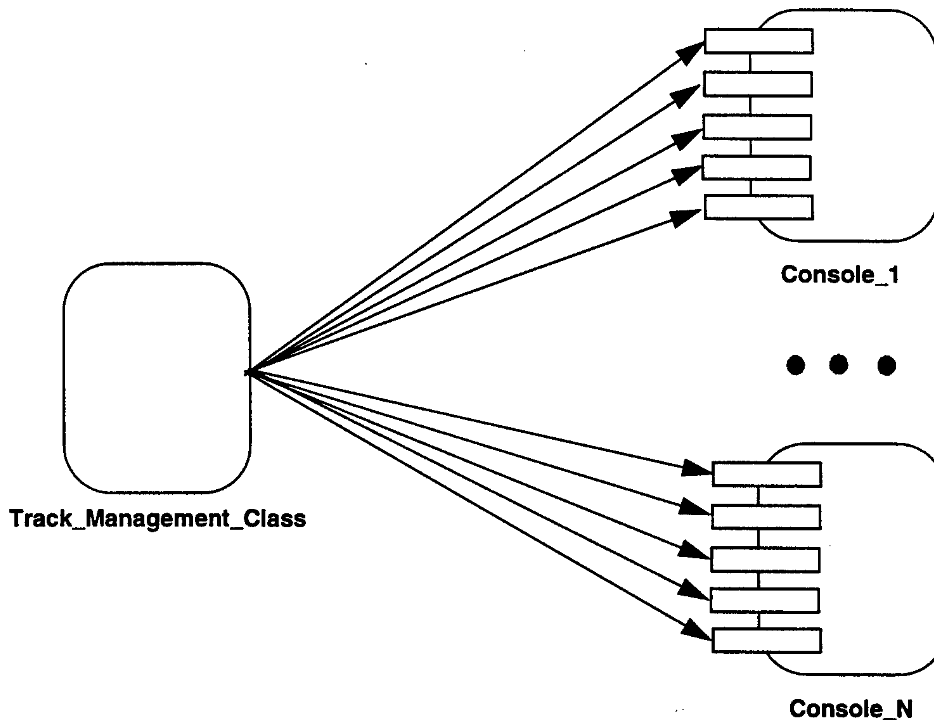


Figure 43: Using Methods to Distribute Track Data

An alternative would be to overload the information that is distributed via a method invocation. For example, there could be one method that would be called *Track Data*. As part of the data specification, there would be a code indicating the function (such as a new track) or a track update (similar to a variant record). We prefer not to use this approach because it unnec-

essarily hides the purpose of the method. In other words, the semantics of the method would not be defined by the name of the method, but rather, by the data that are transferred in the method.

4.6.8.3.2 Use of Event Channels

For the use of CORBA event channels, we would have the following:

- Each instance of data, such as information about a new track or a drop track, would be distributed through a separate event channel. Separate event channels are needed because of the different syntactic structure of data passed over the channel.
- A number of models for distribution and receipt of events are possible. For example, we could have a push producer and a pull consumer.

The resulting design would appear (for the case of the push producer and pull consumer) as shown in Figure 44.

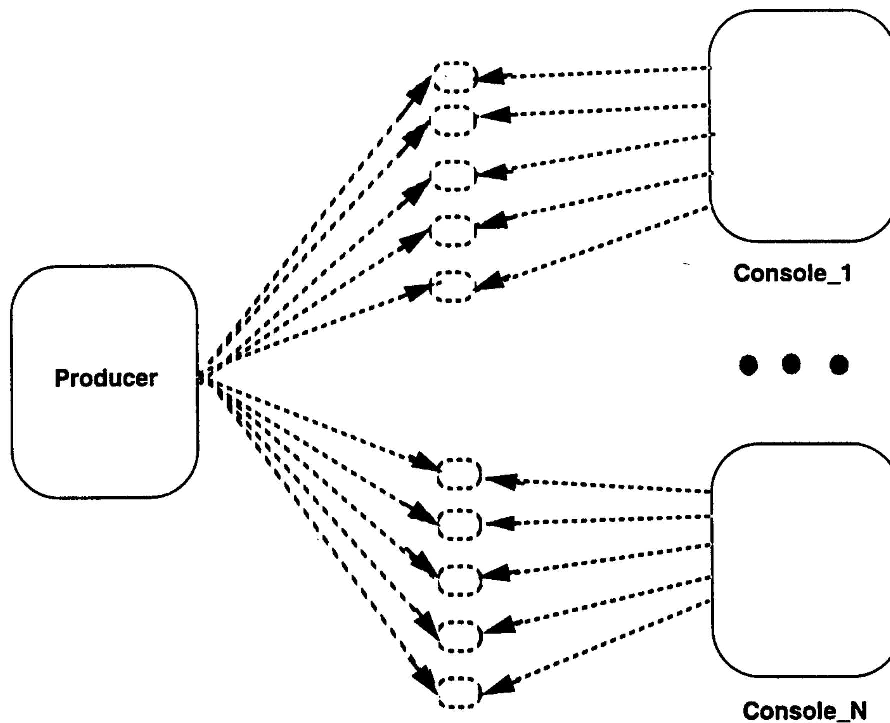


Figure 44: Use of CORBA Event Channels for Distribution of Track Data

4.6.8.3.3 Discussion

The methods and event channels for data distribution are compared and summarized in Table 4.

	Methods	Event Channels
Advantage	• Simplicity	• Can exploit asynchronous interaction • Possibly more flexible, e.g., ease of connectivity
Disadvantage	• Forces synchronization between sender and receiver of track data	• Doubles the number of remote procedure calls • Lack of semantics defined by CORBA • Longer time to detect a console failure

Table 4: Comparing Methods and Event Channels for Distribution of Track Data

Based on the above information, we make the following design consideration:

Design Consideration C-11
The distribution of track data will be achieved through the use of method invocations, not event channels.

Two major considerations in making the above choice were performance and fault tolerance. In addition, we are struck by the lack of semantics in the CORBA documentation about event-management services. For example, all semantics of quality of service are left to the implementor.

Note that in assessing both the use of method invocation and asynchronous events, we assumed one method (or event) for each type of data being communicated. That is, there would be one method (or event channel) for the update of information on a track, one method (or event channel) for an indication that a track had to be dropped, and so on. In both cases, we could encapsulate the track information (update or drop, for example) into one method (or event channel) that could be called *Track Notification*. Such a choice would hide the actual function by encapsulating the information in some data structure. Doing so, however, may complicate the design, and it is not clear that there would be a benefit in terms of performance.

It is possible to further refine the track management in a number of ways. Of special concern was the large number of remote procedure calls necessary to distribute track data. Hence, perhaps one would like to be able to minimize the number of remote procedure calls (which is how methods communicate). One alternative is to buffer track data on a console basis before invoking the method to distribute the track data. This would decrease the number of method invocations by the amount of track-data buffering that can be achieved. However, there could

be additional latency because the buffer that contains the track data would be delayed in its distribution. The refined design is silent with respect to this issue, and it would be possible to incorporate track-data buffering. However, note that there is a question of where the buffering is done. If it is done in track management, it assumes knowledge of the sector topology which could change. An alternative is for the buffering to be performed in airspace management.

Another alternative is for track management to distribute data to consoles directly, without routing by airspace management. Airspace management would provide console geometry to track management on which track data are distributed. Recall that the distribution of track data is based on the geometry associated with a console. Thus, it may be possible for a console to inform the track manager of its geometry, allowing that manager to distribute data to the appropriate consoles. This is a form of caching console_ids for a given track identifier. Such a choice would certainly decrease the role of the airspace management object and, for this reason, was not chosen.

Both of the above choices are expected to provide increased performance over the refined design. However, they do so at a cost. For example, the collapse of function into track management would remove some of the simplicity present in the refined design. Furthermore, it needs to be demonstrated that the refined design will not meet its performance requirements to warrant further change to the design. We would prefer, as a general consideration, to meet the performance and fault-tolerance requirements, while at the same time achieving a simple design which can be extended.

4.6.9 Flight-Plan Management

The purpose of flight-plan management is to encapsulate information about flight plans. This is achieved in the initial design and will be retained in the refinement of the design. There is only one object for the flight-plan management function.

Flight-plan management maintains the following state data:

- objects registered to receive flight plans and the type of flight plans requested
- a flight-plan database with the following state data for each flight plan:
  - aircraft ID
  - aircraft data
  - beacon code
  - speed
  - coordination fix
  - coordination time
  - assigned altitude

- requested altitude
- route

The methods associated with the flight-plan management class include the following:

- Initialize: performs initialization for flight-plan management
- Flight plan register: instructs flight-plan management to provide flight plans to an object
- Flight plan de-register: instructs flight-plan management to stop providing flight plans to an object
- Correlate flight plan: associates a track with a flight plan
- De-correlate flight plan: disassociates a track with a flight plan
- Modify flight-plan data
- Get flight plans for region: returns a list of flight plans for the specified region
- Flight plan update: creates new or updated flight plan from an external source
- Flight plan purge: provides notification from an external source that a flight plan or set of flight plans are no longer valid

The above specification is basically a refinement of the initial flight-plan management presented in Section 4.5.10. In terms of our notation, the flight-plan management class would appear as shown in Figure 45:

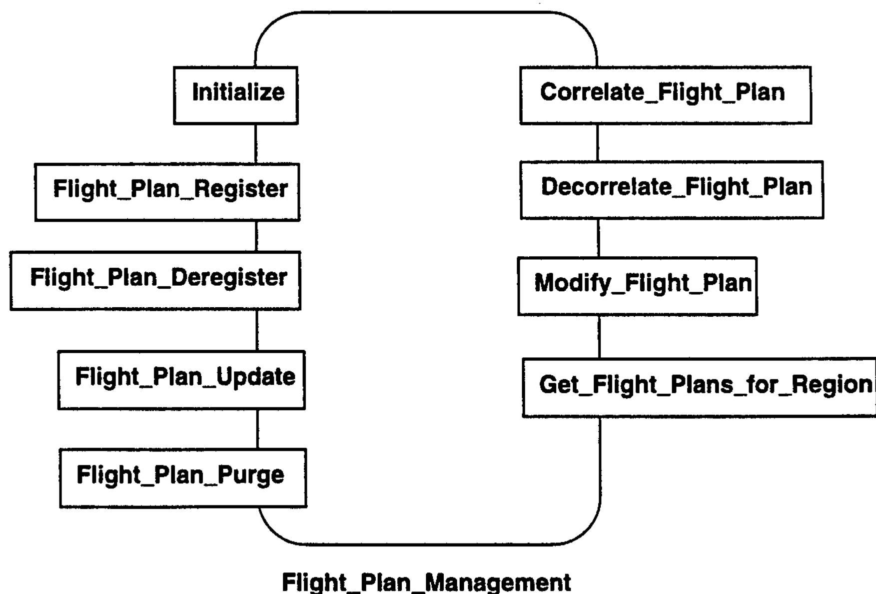


Figure 45: Refined Object Specification for Flight-Plan Management

4.6.10 Flight-Plan Data

The refinement of flight plan data is similar to the refinement for track data. Thus, we first make the following design consideration:

> **Design Consideration C-12**
> Flight-plan data will be distributed only to those consoles that require information about a particular flight.

Similarly, we make the following decision about the way in which flight-plan data will be managed:

> **Design Consideration C-13**
> Flight data will be managed in a centralized manner.

Finally, we make the following decision about how the flight-plan data are distributed:

> **Design Consideration C-14**
> The distribution of flight-plan data will be achieved through the use of method invocations, not event channels.

4.6.11 Flight Plan/Track Correlation

The purpose of the correlation object is to correlate flight plans with track data. There is only one object for correlation function.

The flight plan/track correlation object maintains the following state data:

- A flight plan database containing uncorrelated flight plans
- A track database of uncorrelated tracks

The methods associated with correlation processing include the following:

- Initialize: performs initialization for the correlation
- Track update: creates new track or update of an existing track
- Drop track: track that has been dropped
- Purge tracks: all tracks dropped
- Flight plan update: creates a new flight plan or an update for an existing flight plan
- Delete flight plan: indicates that a flight plan needs to be deleted
- Purge all flight plans: indicates that all flight plans are no longer valid

In terms of our notation, the correlation processing object appears as shown in Figure 46.

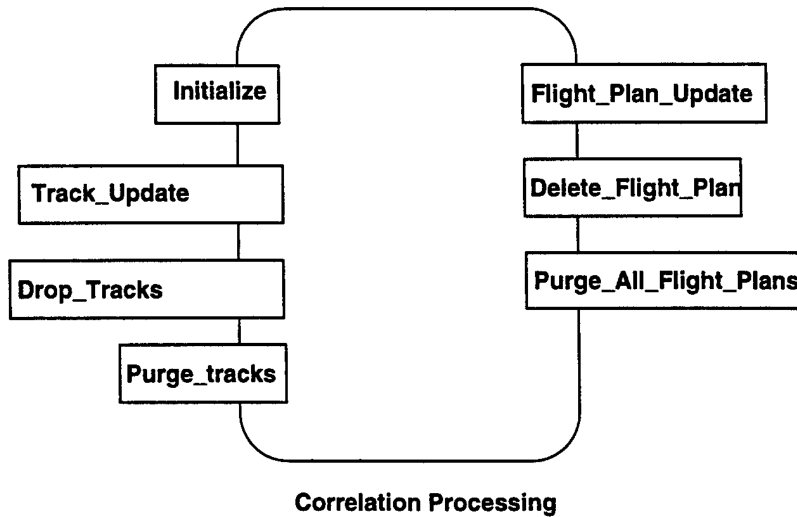


Figure 46: Object Specification for Correlation Processing

4.6.12 Summary of Refined Design

4.6.12.1 Objects and Their Interaction

In Figure 47, we present the result of refining the initial design.

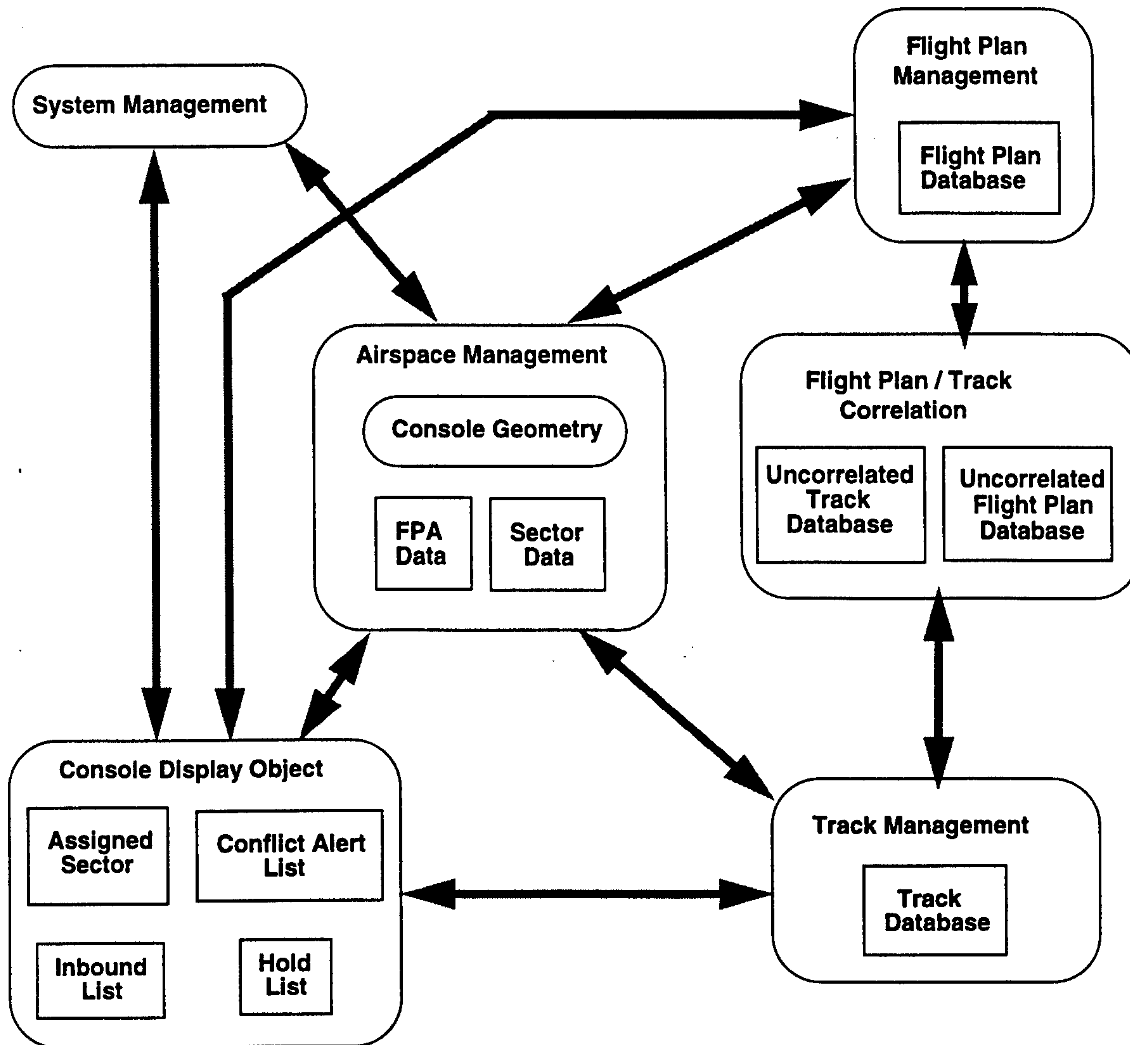


Figure 47: Refined CORBA Design

The shaded arrows in Figure 47 denote interaction between the components indicated. Note that much of the initial object character of the design is no longer present in Figure 47. The principal reasons for this are as follows:

- Tracks and flight plans, which contained the largest number of objects, are now represented as databases. This is because of the need for dealing with many-to-many interactions (such as correlation), as well as performance and fault-tolerant considerations.
- Data lists, such as inbound and hold lists, are simply attributes of a sector. Hence, their treatment as an object was not warranted.
- There is a resulting encapsulation of function through the use of objects.

4.6.12.2 Mapping the Refined Design onto Hardware

It is interesting to map the resulting CORBA design onto hardware. We shall now consider two different examples of possible mappings.

The first mapping to consider is termed a centralized mapping. In this case, the current host functionality associated with track management, flight-plan management, and track/flight plan correlation is mapped onto a central hardware component. This is shown in Figure 48.

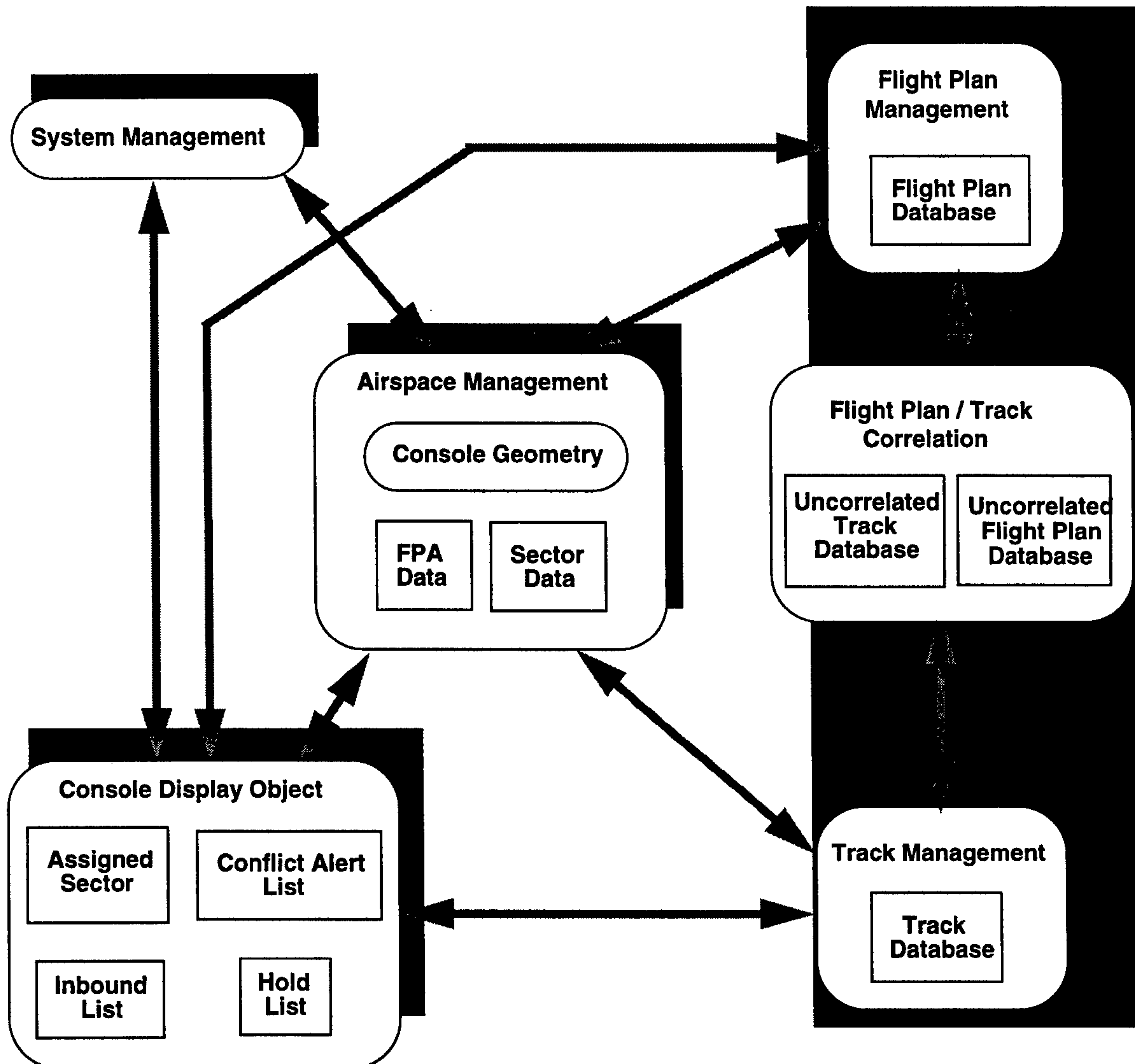


Figure 48: Centralized Hardware Mapping of CORBA Design

The second mapping to consider is that in which the major host-related components (track management, flight-plan management, and track/flight plan correlation) are each mapped to an individual processor. This represents a distributed mapping onto the hardware components and is shown in Figure 49.

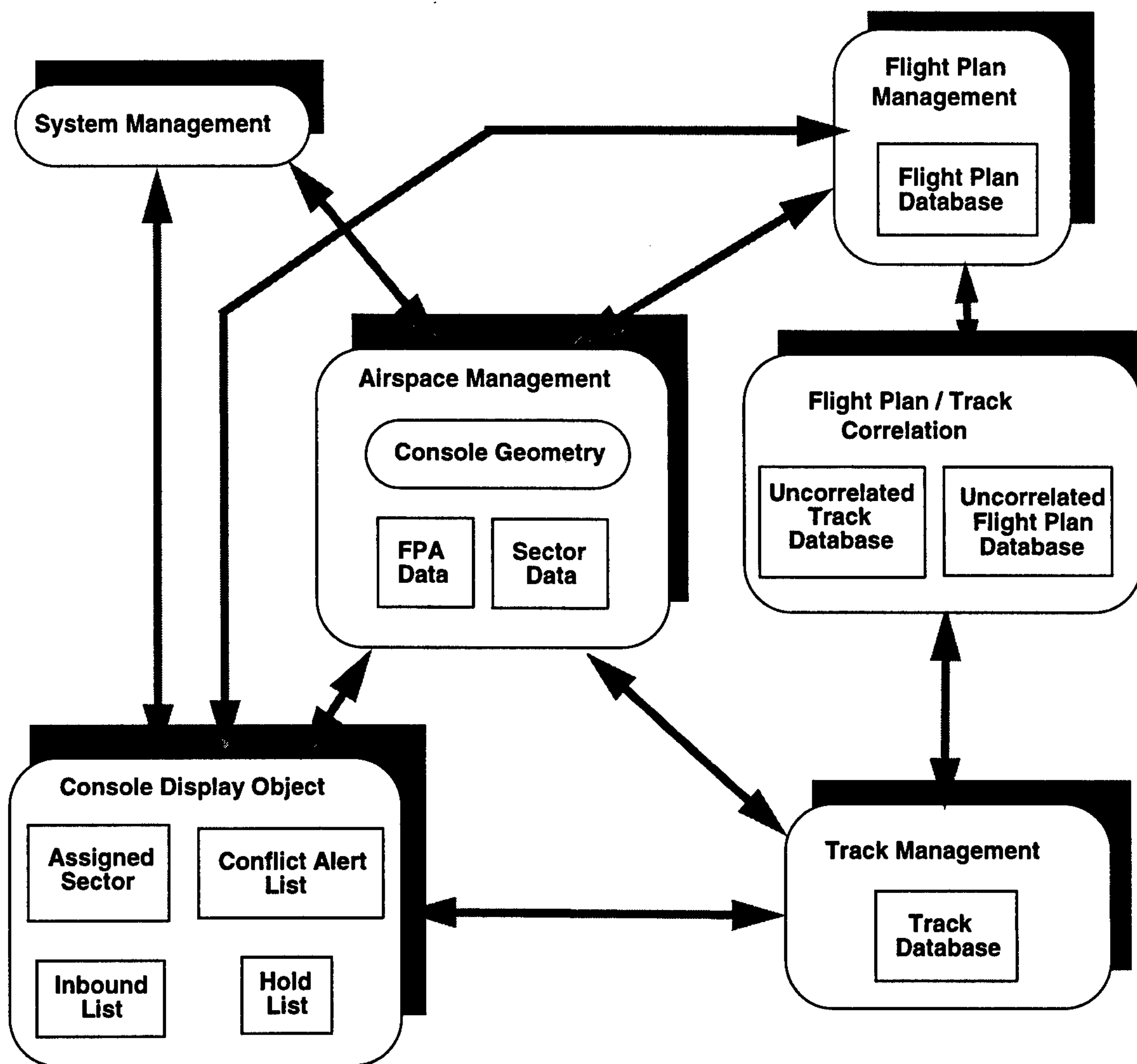


Figure 49: Distributed Mapping of CORBA Design

Although the hardware mappings satisfy functional requirements, they have different performance characteristics. For example, in the case of the centralized mapping, it is possible to use local interprocess communication since the processes are assumed to reside on the same machine. However, in the case of the distributed mapping, the communication becomes non-

local in nature. Of course, the lack of close coupling in the distributed mapping may be advantageous from a system-evolution perspective.

4.6.12.3 Sample Data Flows

4.6.12.3.1 Track Data

Figure 50 shows the sequence of events to update a display console with new track data.

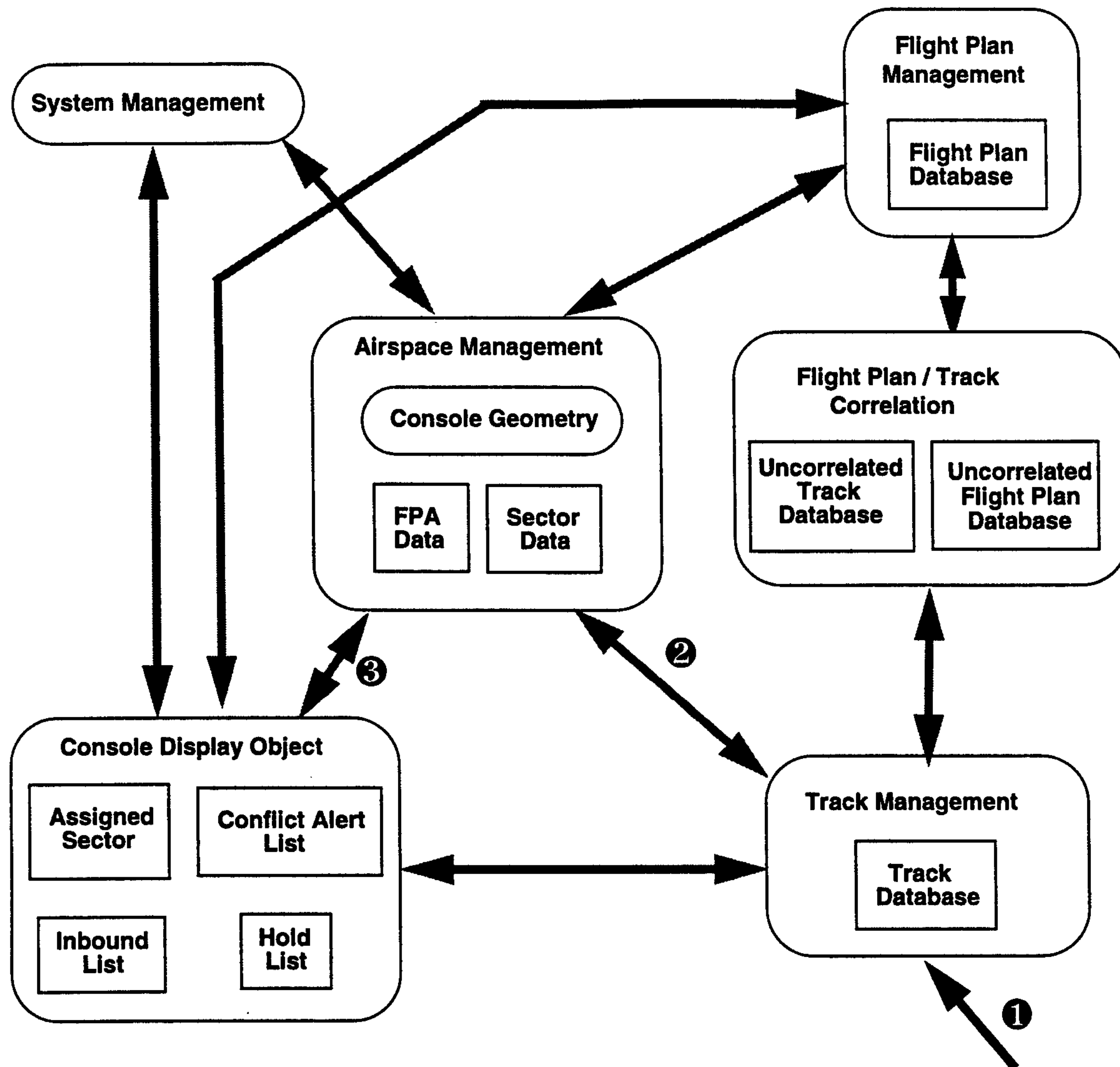


Figure 50: Refined Design for Track-Update Data Flow

The sequence of events for the track update is described below (we are neglecting track correlation as in the previous example):

1. Receive new track data: Track data are assumed to be provided from an external system, such as a controller for surveillance processing. These data are then sent to track management.
2. Send track data to airspace management: The track data are sent to airspace management, which is responsible for routing the data to one or more display consoles.
3. Route track data to console(s): Airspace management maintains information about the geometry that a display console is controlling. When track information arrives, we assume it has the current track position. this information is used by airspace management to map the track position onto one or more consoles. Then, the method is invoked on the appropriate consoles to provide them with new track data.

Note the simplicity of track-data distribution compared to the initial design (see Figure 29 and the ensuing discussion). One of the reasons for the simplicity is the simplification of the initial design with respect to the number of objects maintained.

4.6.12.3.2 Flight-Plan Data

The treatment for flight-plan data is similar to the earlier case. That is, it is necessary for information about a new flight to be distributed. In terms of the refined CORBA design, Figure 51 illustrates the distribution of data.

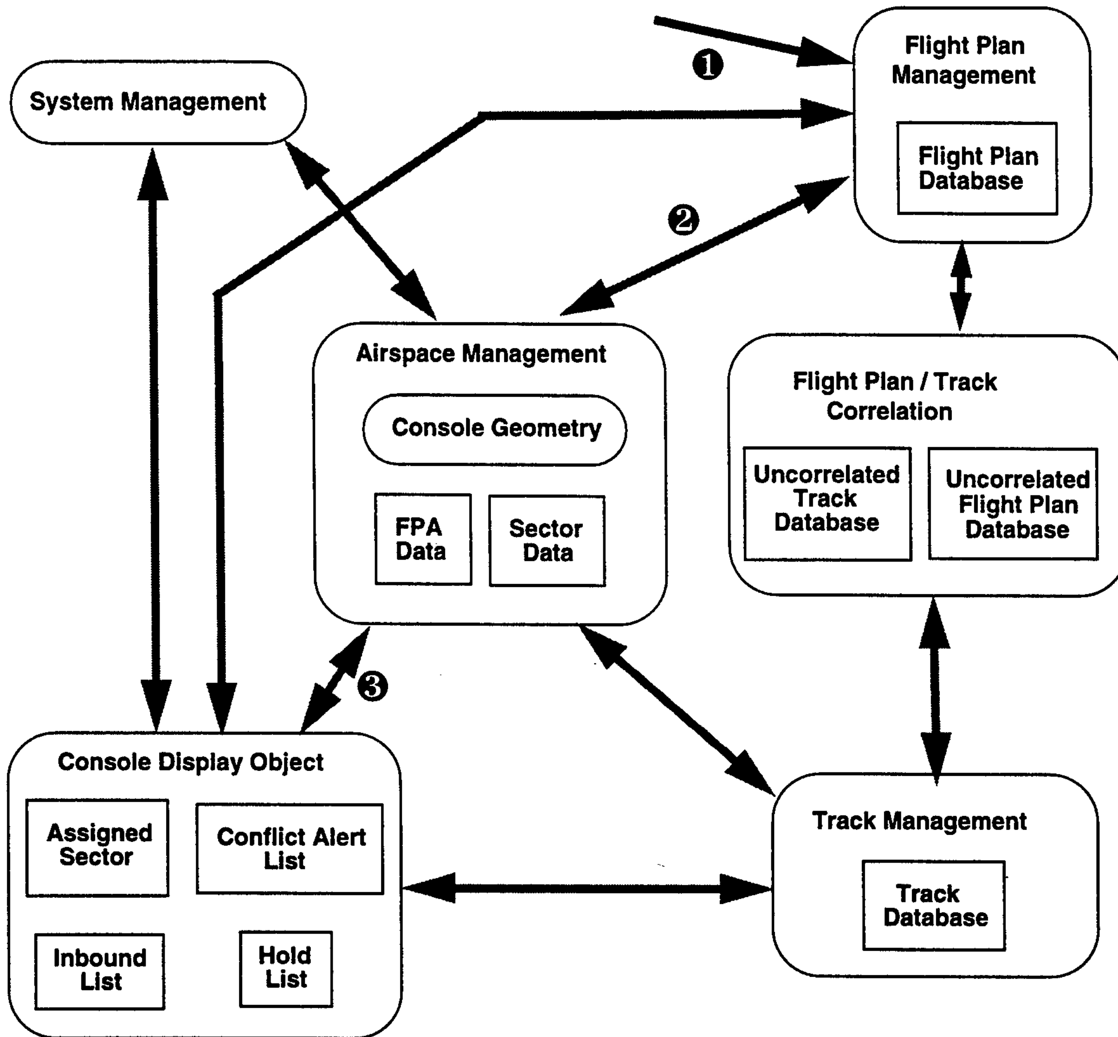


Figure 51: Refined Design for Data Flow of Flight-Plan Update

The sequence of steps in the flight plan update is described below (again, we neglect correlation processing):

1. New flight-plan data: We assume that there is new information abut a flight plan and that this information is provided to flight-plan management.
2. Send data to airspace management: The flight-plan data are sent to airspace management, which is responsible for routing the data to one or more display consoles.

3. Route flight-plan information to console(s): Airspace management maintains information about the geometry that a display console is controlling. When flight-plan information arrives, airspace management maps the flight plan onto one or more consoles. Then, the method is invoked on the appropriate consoles to provide them with new flight-plan data.

As in the case for revised track-data updates, the distribution of flight-plan data in the revised design is simpler than in the initial design. There is also symmetry between the distribution of flight-plan and track data.

4.6.12.3.3 Sector Combination

The case where two sectors are combined was one of the motivating points for this report. A description of the data flow for sector combination in the refined design is presented in Figure 52.

The sequence of steps in the example is described below:

1. Operator request: A system management operator invokes the request to combine the specified sectors.
2. Send request to airspace management: After the request is validated at system management, a method is invoked on airspace management to combine the desired sectors.
3. Make requests of consoles: Airspace management maintains knowledge of what sectors are currently mapped to a particular console. A request to combine two sectors means that console geometry will be changed. Hence, airspace management invokes methods on the affected consoles, indicating that they should change their configuration.
4. Route data to new console configuration: After the sectors have been combined, it is necessary to route new data to the appropriate console. This routing information is known by airspace management. Hence, when new data (such as track or flight-plan data) arrive, these data will be routed to the newly combined consoles, as appropriate.

As in the examples for the distribution of track and flight-plan data, the refined design is simpler for the case where sectors are combined.

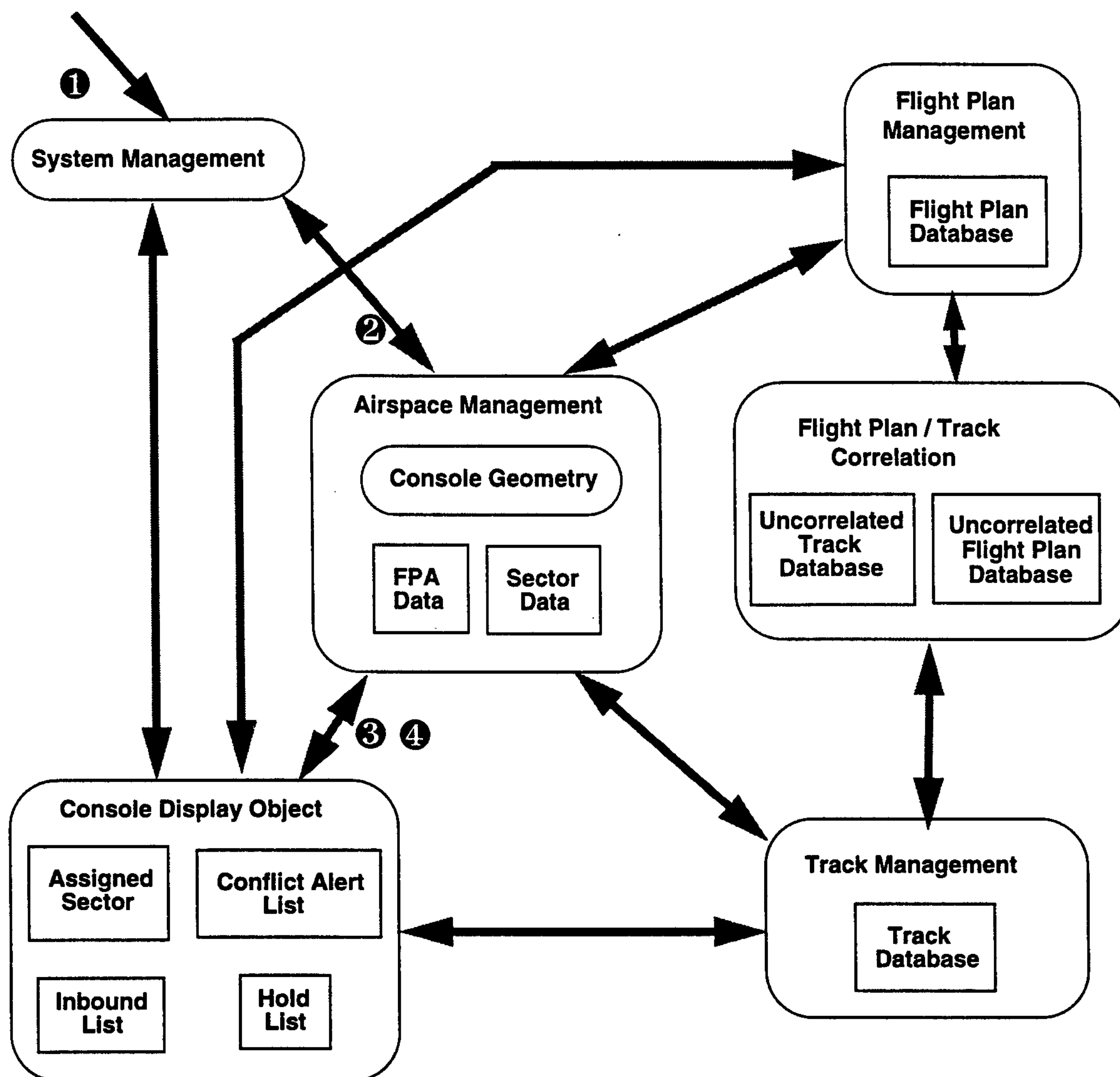


Figure 52: Refined Design for Sector Combination

4.6.12.3.4 Console Failure

The final example we shall present is that where a console fails such that no objects can communicate with the console. This case is shown in Figure 53.

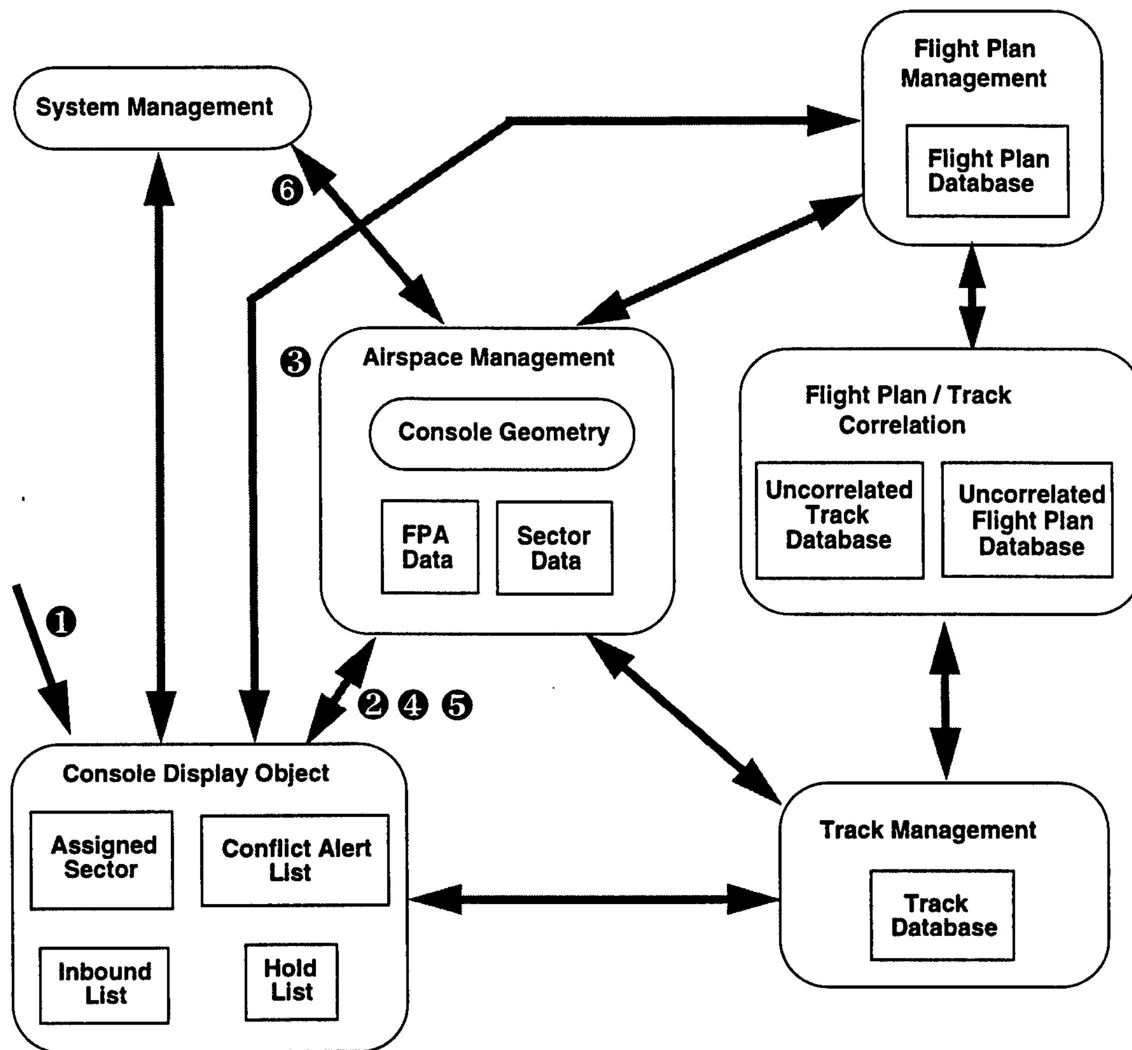


Figure 53: Refined Design For Console Failure

The sequence of events for the above figure is described below:

1. Console fails: An active display console undergoes a failure such that it is incapable of sending or receiving any data.
2. Failure detected: An object attempts to invoke a method on a console. Assume, for the present case, that the failure is detected by airspace management invoking a method on a particular sector object.

3. Airspace management notification: Due to the central role that airspace management plays, it will need to be notified of a failure in a console. In this case, we are assuming that airspace management detects the failure, but it could be detected by other objects. In either case, airspace management must be notified of the failure.
4. Assign sector to new console: When the console failed, we assume that it was controlling some active sector in the En Route center. To initiate error recovery, it is necessary to assign the sector to a new console.
5. Update new console: It is necessary to initiate recovery for the new console so that it has information about the state of tracks, flight plans, and associated data.
6. Notify system management: The last step in the recovery process is for airspace management to notify system management that the failure has been resolved and a new console is now controlling the airspace that had been associated with the failed console.

This example illustrates the typical sequence of operations that would have to take place in the event of a display-console failure. Basically, a new console must be brought up, and then repopulated with data for the failed console. The function is similar to that presented in the initial design, only here it has been somewhat simplified.

4.7 Implementation Concerns

The suitability of a given technology is a function of the implementations of that technology. During the course of this work, some issues about implementation performance were raised. To resolve these issues, it is necessary to have information specific to a given implementation. Some of this information is described below:

- General performance characteristics: For example, what is the time to invoke a method as a function of the amount of data transferred in the method? Information such as this usually falls under the term of feature benchmarks.
- Event channels: In the CORBA documentation, it states that quality of service for event channels is implementation defined. Hence, one would need this information for a particular implementation.
- Failure detection: A fundamental question is the amount of time it takes to detect a failure of a method invocation. The amount of time is important for fault-tolerant considerations.[1]

1. If the internet protocol mapping (such as TCP) is being used to transport CORBA method invocation data, the TCP keep-alive timer may determine the time to detect a failure. Traditionally, the values of the TCP keep-alive timer have been quite large (say 30 seconds or more). However, there may be implementation dependencies that decrease the time until a method failure is detected.

There are a number of other implementation-specific concerns that can be studied. Most of these can be represented as classes of benchmarks and will not be pursued here.

4.8 Summary of CORBA Design

As a means of summary, Table 5 compares the initial CORBA design to the refined design.

Object Class	Initial Design	Refined Design
System Management	Serves to encapsulate system state data	Same as initial
Airspace Management	Serves to encapsulate airspace objects	Same as initial
FPA	One object per FPA	Attribute of a sector, not an object
Sector	One object per sector	Data managed by airspace management
Console	One object per console	Same as initial
Inbound List	One object per inbound list per sector	Attribute of a sector, not an object
Hold List	One object per hold list per sector	Attribute of a sector, not an object
Conflict-Alert list	One object per conflict-alert list per sector	Attribute of a sector, not an object
Track Management	Serves to encapsulate track objects	Same as initial
Track	One object per track	Collapse track objects into a track database
Flight-Plan Management	Serves to encapsulate flight-plan objects	Same as initial
Flight Plan	One object per flight plan	Collapse flight-plan objects into a flight-plan database

Table 5: Summary of CORBA Design-Object Characteristics

It is important to emphasize that the rationale for the resulting refined design was based on performance and fault-tolerant considerations. An additional characteristic is the resulting simplicity of the refined design.

5 POSIX.21 Approach

5.1 IEEE POSIX P1003.21

5.1.1 Background

The Institute of Electrical and Electronics Engineers (IEEE) is involved in the development of standards in a number of computer-related fields. Of particular relevance to this report are the application program interface standards which are managed by the Portable Applications Standards Committee (PASC). These are principally defined in terms of C and Ada language bindings. Among the standards that are developed by PASC, one major subset is the family of well-known POSIX (Portable Operating System Interface) standards.

The 1003.21 draft standard is presented as a language-independent specification (LIS). It is based on operations on abstract data types and does not include programming language constructs. Hence, LIS allows one to specify the interface *semantics* without reference to how those semantics will be *implemented* in a language binding (such as Ada or C). Given a language-independent specification, it is possible to derive language bindings.

The process by which the IEEE accepts a specification is outlined below:

1. A group of people recognize a need for a standard specifying some functionality related to some domain.
2. The group develops a project authorization request (PAR) which is reviewed and approved by the PASC sponsor executive committee.
3. The group develops a standard for the intended functionality.
4. The standard is balloted by members of the IEEE and affiliated societies. The only restriction on voting is that an individual be a member of the IEEE or an affiliated society. Other votes are accepted, and the working group is encouraged (but not required) to address such comments.
5. The working group attempts to resolve ballot comments, reaching a 75% consensus among the ballot group.

When consensus is reached, the standard is officially approved by the IEEE. POSIX standards are *fast-tracked* to then become ANSI (American National Standards Institute) and ISO (International Organization for Standardization) standards.

5.1.2 Architectural Overview

In an architectural context, POSIX.21 is a component of an operating system. More specifically, because it resides in part of a family of POSIX standards, it would be a part of a POSIX operating system interface. We represent this as shown in Figure 54.

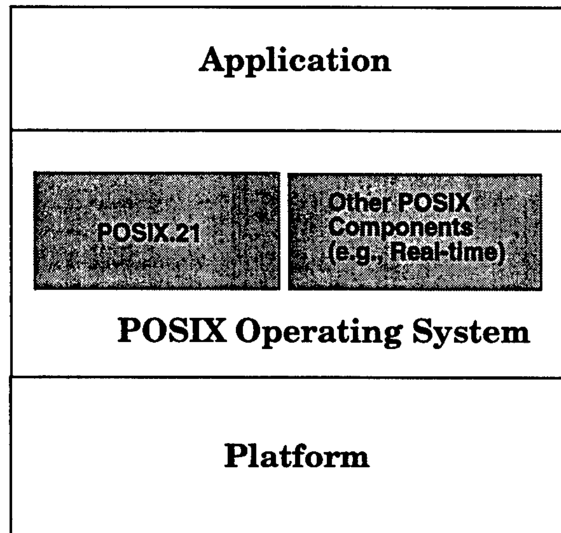


Figure 54: Architectural Context for POSIX.21

The POSIX standards are an organizational subset of those managed by the PASC. Hence, PASC standards may be appropriate for consideration in an acquisition. Examples of these standards include interfaces for directory services and ASN.1.

5.1.3 Real-Time Considerations

The IEEE draft standard 1003.21 (Real-time Distributed Systems Communication) has been developed to meet the needs of applications in the real-time computing domain, in particular distributed systems. Features included to meet real-time needs include

- buffer management: The 1003.21 standard supports buffer management where the buffers reside in *implementation* space. This capability is expected to permit *zero-copy* implementations, allowing for shorter and more deterministic latencies for message transfers.
- asynchronous and synchronous interaction with the implementation: Most of the operations are specified in either an asynchronous or synchronous manner. This permits the development of flexible applications based on their communication needs.
- bounding blocking in the implementation: Most of the operations have an associated time-

out. The use of a time-out will bound the delay that can occur while the implementation is executing a requested operation. This helps to eliminate problems in which the implementation can become blocked while providing service to an application.

- message priorities: Each message may have an associated message priority. The implementation must service high-priority messages before servicing low-priority messages. This approach eliminates well-known problems associated with first-in-first-out (FIFO) queueing of messages (such as priority inversion). Two representations of priority are specified. One is based on a simple integer representation, the other on a deadline representation.
- message labels: A message may have an associated *message label* that is transferred with the message. The use of message labels is an efficient model for applications, since they need not know the set of all intended receivers. An application may also register to receive messages of a specific type, allowing for message filtering by the implementation. The semantics of operations on message labels do not require or imply a particular implementation approach. This approach is analogous to a publisher-subscriber model where the negotiated element is the message label that is published and subscribed by applications.
- implementation protocol: A definition of a protocol *header* for use between implementations is defined. This protocol is needed to support interoperability of message priority, type, and format.

These are some of the more important considerations that the 1003.21 standard includes with respect to real-time systems.

5.1.4 Communication Mechanisms

To understand the communication mechanisms provided by the 1003.21 standard, we begin by briefly noting the basic interface components specified in the standard. These include

- endpoints: An endpoint is an object that is created and maintained by the implementation. It is used by applications for sending and receiving messages, and by the implementation for identifying the source and destination of the messages.
- messages: A message is a sequence of octets that is transferred from a local endpoint to one or more remote endpoints. The semantics associated with the message data are defined entirely by an application.
- buffers: A buffer is an area in memory used to store application messages. The use of buffers is expected to permit *zero-copy* operations–an important optimization in real-time considerations. An application can also allocate buffers from application space and use those buffers in data-transfer operations. However, this choice may result in a copy operation, making it less efficient.

- logical names: A logical name may be used to identify one or possibly more (in the case of a multicast group) endpoints.
- multicast groups: A multicast group is defined to be a set of endpoints. Multicast groups can be created and deleted by applications, and endpoints may join or leave a group.

The 1003.21 standard supports a number of communication mechanisms. The communication takes place between an endpoint and either a destination endpoint or an identifier that is used to denote a set of endpoints. The basic mechanisms include

- one-to-one (unicast): transfer of a message from a source endpoint to a destination endpoint. The destination endpoint is not required to be within the same process as the source endpoint.
- one-to-many (multicast): transfer of a message from a source endpoint to a set of known destination endpoints, denoted by a multicast group.
- one-to-any: transfer of a message from a source endpoint to a set of destination endpoints that have registered to receive the message, but are unknown to the sender. The messages are denoted by a message label which an application can register (or deregister) to receive. The sender does not know who receives the message.
- one-to-all (broadcast): transfer of a message from a source endpoint to all other possible destination endpoints. The scope of *all* is determined by the implementation.

Associated with a communication mechanism, there can be various quality-of-service attributes. Two important attributes are

- priority: a requirement on the implementation to provide a queueing policy for sending and receiving a message. A message may have a message priority of either an integer value or a deadline.
- reliability: a requirement on the implementation to deliver the message without failure.

The relation between the communication mechanism and quality of service is summarized as follows:

- The priority of a message can be included with any message.
- A message label can be included with any message.
- Reliability can be specified for one-to-one and multicast communication models.

5.1.5 Additional Services

The following topics are also included in the 1003.21 standard:

- *Initialization and termination:* An operation for the implementation initialization is provided by the standard. This initialization must be performed before any other operation is

initiated. Termination semantics are also defined in the 1003.21 standard. There are two different cases in which the termination semantics are necessary. The first is the case in which the application is terminated by some external agent, such as another process. The second is the case in which an application-schedulable unit (either process or threads within the scope of the process) voluntarily terminates. In the second case, two modes of termination are provided: *graceful* and *abrupt*. As an example of the termination semantics, if a thread has created an endpoint and the thread is terminated, the resources associated with the endpoint remain available to other threads.

- *Event management*: There are occasions where an asynchronous event may be detected and raised by an implementation. The 1003.21 standard provides a a way for events to be queued (in a FIFO manner) for the endpoint with which the event is associated. An example of a case where an event is raised is if a remote endpoint has terminated a connection.
- *Directory services:* A lightweight interface for directory services is included in the standard. The interface permits two views of the underlying naming model. In the first case, logical names are emphasized; for example, it is possible to obtain a local identifier for a remote endpoint, specified by its logical name. In the second case, the emphasis is on a protocol-dependent address. Here, for example, an application can define a local identifier for a specified protocol-dependent address.
- *Protocol management and mappings:* The intent of the standard is to make the interface as protocol independent as possible. In practice, however, this may not be possible because of differences in underlying (transport) protocols. The term *protocol mappings* refers to the ability to get and set protocol parameters for a specified endpoint, as well as to obtain visibility of protocol-specific operations. The standard provides protocol support for TCP (transmission control protocol), IP (Internet protocol), and UDP (universal datagram protocol).
- *Connection management:* It was recognized that application developers, who are familiar with connection-oriented data transfer, would need the ability to use such a model in a real-time system. Hence, the standard includes a typical interface for operations such as opening a connection, rejecting a connection request, and so forth.

As noted earlier, because 1003.21 is a part of the POSIX family of standards, these other interfaces would be available. This includes interfaces for a broad range of features. In this report, our focus is on real-time communication, a subset of general real-time issues. Other POSIX standards address aspects of general real-time features, including the following:

- signals
- semaphores
- memory locking and mapping
- process and thread scheduling

- clocks and timers
- interprocess message queues

Both C and Ada language bindings are defined for the above features. There are also pre-defined *profiles* (a collection of one or more standards and their options), one of which is for the real-time domain.

5.2 Presentation of Design Information

Much of the design information presented in this report is in the form of diagrams. We have used these diagrams because they are simple and relatively easy to understand. They also offer intuitive semantics that help the reader to understand the meaning of the diagram.

The diagrams are based on the basic concepts defined in the 1003.21 draft standard, such as endpoints and messages. We also include a means to encapsulate an endpoint in a logical structure. Such a structure could be a process, although we do not require this.[1] We also permit multiple levels of encapsulation, although we emphasize that the structures used to encapsulate basic components (such as endpoints) *do not* imply any mapping to underlying hardware. Instead, the encapsulations are purely logical in nature, and multiple mappings of an encapsulation structure onto hardware can exist.

Figure 55 illustrates the design notation for communication between a system-management entity for sector reconfiguration management and a local management agent.

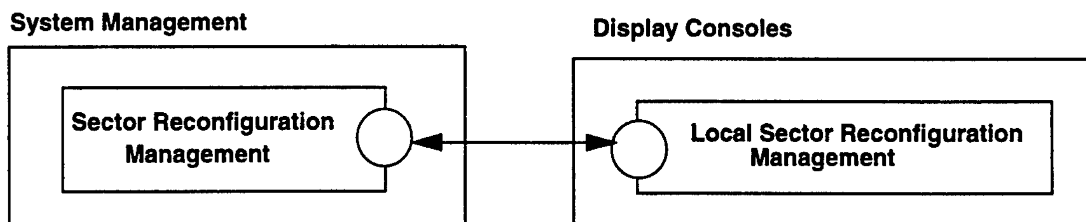


Figure 55: Sample POSIX.21 Design Notation

1. In the 1003.21 draft standard, the operations are defined within the scope of a process. This may be viewed as a basic encapsulation mechanism. However, the 1003.21 draft standard also includes semantics for operations in the context of a thread (or task) when necessary. For example, if a thread creates an endpoint and the thread is deleted, it is necessary to define what happens to the endpoint that was created by the thread.

The following symbols are used in the diagram:

- *Rectangles* denote encapsulation. For example, there is an entity called System Management that includes an entity called Sector Reconfiguration Management.
- *Circles associated with a rectangle* denote an endpoint.
- *Arrows* indicate a message transfer. The direction associated with the arrow denotes the direction of data transfer. In Figure 55, messages are sent to and from Sector Reconfiguration Management (in System Management) and Local Sector Reconfiguration Management (in Display Console).

It may sometimes be necessary to indicate whether a message is transferred in a reliable manner. We will denote reliable and unreliable message transfers by the use of solid and dashed lines, respectively. In addition, it may also be necessary to denote the way in which a message is transferred as either multicast, broadcast, or as a labeled message (denoted by the use of the letter M, B, and L, respectively associated with the arrow). If an arrow is not denoted with any of the above, we assume it will be sent in a unicast manner.

The above refinements are presented in Figure 56:

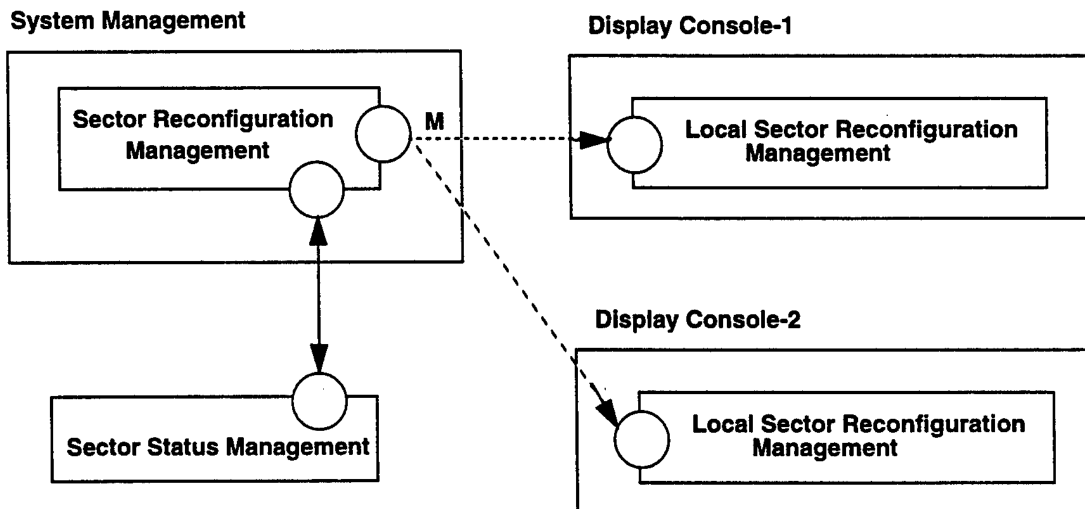


Figure 56: Additional POSIX.21 Notational Devices

In Figure 56, note the following:

- Sector Reconfiguration Management multicasts a message, in an unreliable manner (denoted by based lines) to the two display consoles.

- Sector Reconfiguration Management also exchanges messages, in a reliable manner (denoted by solid line) with an entity called Sector State Management. The lack of a label on the arrow denotes that messages are exchanged in a unicast manner.

There is clearly a simplicity in presenting design information in a graphical manner, as illustrated above. This approach will be used for expressing the POSIX.21-based design.

5.3 Basic Design Issues

5.3.1 General Considerations

The design process for a POSIX.21-based system is oriented toward the distribution of data, via messages, among system components. Some of the main issues that need to be addressed in the software context, include the following:

- What is the mapping of system functionality onto software processes?
- What is the allocation of data onto messages, and how are they sent/received?
- What is the allocation of endpoints within a process?
- What is the relationship between endpoints and threads?

Each of the above points raises major issues that require consideration in a full-scale design and development effort. It is not our intent to go into detail about how one would address issues such as those raised above. However, examples will be presented. To illustrate one of the cases, note that the relation between endpoints and threads is coupled to the question of endpoint-message relationships. From the perspective of thread considerations, one could have the following:

- one task per message (sent and/or received): This approach leads to a design that is highly message-specific in nature. If there were a large number of messages, there would be a large number of tasks.
- one task per communication model (i.e., one task and endpoint for sending and receiving broadcast messages, one for unicast, etc.). This approach focuses on the intended receivers of the message.
- one task for reliable messages and one task for messages that do not have reliability constraints. This approach is oriented toward quality-of-service considerations. Following this line, it is possible for there to be a separate task for high-priority messages.

Other choices are possible. In this report, we are more interested in architectural properties. Hence, it is our intent to refrain from lower level design issues to the extent possible. However,

as we shall see later, it will be necessary to selectively reveal lower levels of design information (such as threads) to perform a design analysis.

5.3.2 Use of Endpoints

An endpoint is the basic communication object defined in the POSIX.21 standard. As such, the degree to which endpoints are used needs to be considered as part of the design decisions. A number of factors need to be considered in the performance aspects of using endpoints.[1] These factors include the following:

- memory required to allocate an endpoint
- time to create and/or delete an endpoint[2]
- management time for endpoints with respect to data transfer operations

For example, the amount of memory allocated to an endpoint is about 180 bytes, which appears reasonable. We conclude the following:

> **Design Consideration P-1**
> The design shall not be overly constrained in the number of endpoints that are used for communication.

The preceding consideration illustrates an important point about the design of real-time systems, particularly those that are distributed in nature. That is, the design is typically influenced by performance factors of the underlying components. Knowledge of such factors early in the design process helps to ensure that overall system-performance characteristics can be achieved.

---

1. An earlier description of this problem was addressed in a limited context. Meyers, B. Craig & Place, Patrick R. H. *The Use of IEEE Draft Standard 1003.21 Real-Time Distributed Systems Communication in an FAA En Route Architecture* (CMU/SEI-Special Report). Pittsburgh, Pa.: Software Engineering Institute, March 1, 1997.

2. Endpoint deletion time is important for mode change considerations.

5.4 Architectural Considerations

5.4.1 Chosen Architecture

The chosen architecture for the POSIX.21 design is similar to that for the CORBA design (see Section 4.5). Thus, we assume the following entities:

- display consoles
- system-management consoles
- airspace management
- flight-data processing
- track-data management

The flight-data processing and track-data management functions are intended to replace the current Host computer system processing. The incorporation of POSIX.21 in the ARTCC architecture is based on the following design consideration:

Design Consideration P-2
We assume that the DSR consoles can contain an implementation of POSIX.21.

To accommodate the above design consideration, we need the following, somewhat stronger statement:

Design Consideration P-3
We assume that any component of the ARTCC, such as flight-data processing, track management, or system management, can contain an implementation of POSIX.21.

The resulting architecture, which includes the above design considerations, appears in Figure 57.

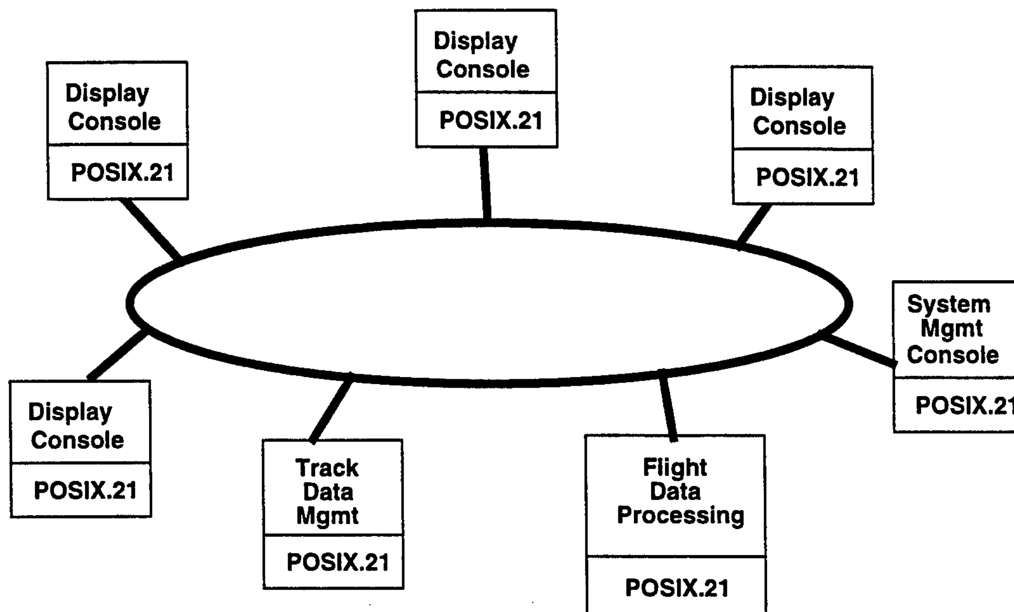


Figure 57: Assumed POSIX.21 Architecture

The important point to note in the above figure is that we are assuming that POSIX.21 is present on all system components. This would require a change to the current DSR to accommodate a POSIX.21 implementation. If POSIX.21 is to be considered seriously for inclusion within an ARTCC in a long-term solution, it should be fully distributed over all necessary components.

The ring connectivity in Figure 57 should be interpreted as logical connectivity. We are aware that the DSR connectivity among consoles is a multi-ring structure, and there is nothing that would prevent maintaining that connectivity in the context of a redeveloped host. In this report, we address the components rather than their physical connectivity.

5.4.2 Migration Considerations

The architecture presented in Figure 57 includes POSIX.21 in all components of the En Route architecture. The current DSR system includes a POSIX.1-conformant implementation. To the extent that a C-language binding to POSIX.21 would be an extension of the POSIX.1 standard, it should be possible to include POSIX.21 in the DSR.

In addition, the current DSR design makes use of communication over sockets (an interface which is defined in the recently approved IEEE 1003.1g C-language binding standard). We would expect that the transition from communication over sockets to a POSIX.21-based model should be relatively straightforward.

5.5 Design

5.5.1 Design Components

The principal components of the POSIX.21 design include the following:

- system management
- airspace management
- track management
- flight-plan management
- display consoles

The function and way in which the above components communicate are discussed in the following subsections.

5.5.1.1 System Management

System management is responsible for the overall management of the system. In the context of sector reconfiguration, we will allocate an endpoint that is used to communicate with consoles for the purpose of managing the sector and FPA assignments. The resulting diagram (Figure 58) shows the configuration of system management.

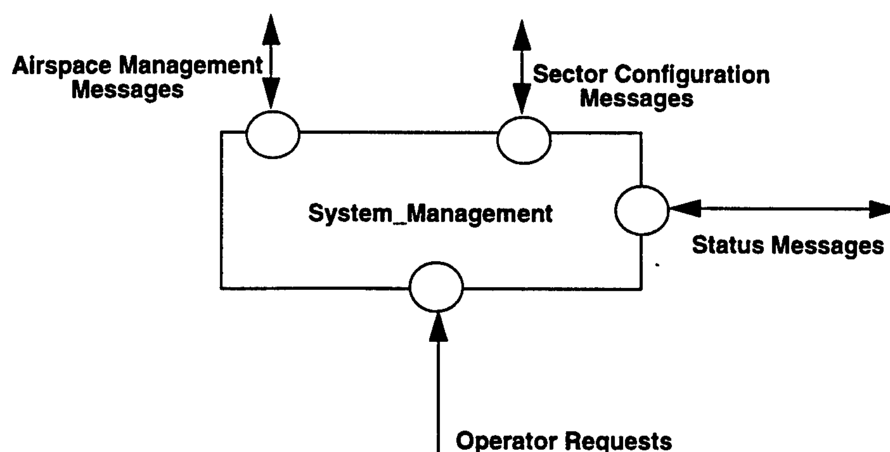


Figure 58: POSIX Design for System Management

There is also an endpoint associated with system management whose purpose is to initiate requests for status of various devices in the system. We assume that there is a response to such requests. The purpose of this is to provide one mechanism for system management to detect a failure in a device, notably a display. Upon recognition of a failure, we expect reconfiguration processing to be initiated. For example, if a console failure is detected, system management could initiate a warm restart of a null console. We will discuss this case in more detail below, particularly the maintenance of state data.

Also shown in Figure 58 is an endpoint for communication with airspace management. The rationale for this is discussed in Section 5.5.1.2.

5.5.1.2 Airspace Management

Airspace management can be viewed as a subfunction of system management. We assume that airspace management would be responsible for maintenance of the following information:

- mapping of sectors onto consoles
- mapping of FPAs onto sectors (and consoles)

The resulting design for airspace management appears in Figure 59:

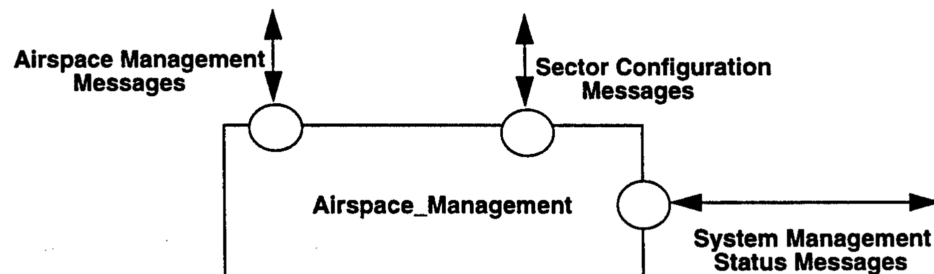


Figure 59: POSIX Design for Airspace Management

There are two endpoints associated with airspace management. One is used for communication with system management. For example, a request to combine sectors or assign a sector to a particular console would be processed through this endpoint. The second endpoint is used for communication with individual consoles for sector (and FPA) configuration messages.

5.5.1.3 Track Management

5.5.1.3.1 Central or Distributed Management

A basic question regarding track data is whether it is managed in a centralized or distributed manner. For reasons similar to that in the discussion of the CORBA design (see Section 4.6.8.2), we make the following design consideration:

> **Design Consideration P-4**
> We assume that track data are managed in a centralized manner.

One major reason for centrally managing tracks is the need to perform track conflict-alert processing, which requires comparison of one track's characteristics versus many other tracks. Another reason is the ability to reconstitute a console in the event of a failure. However, as we will soon see, this is less of a need in the POSIX.21 design because of the method in which tracks are distributed.

5.5.1.3.2 Distribution of Track Information

Given that track data are centrally managed, another basic question concerns the way in which these data will be distributed to consoles. Two choices for this issue are described below:

- Distribute track data only to those consoles that require the data: This was the choice in the CORBA design (see Section 4.6.8.3) where track data were sent only to a console that was unable to display the track.
- Distribute track data to the set of consoles that could be required to display the data.

Based on the above two choices we make the following design consideration:

> **Design Consideration P-5**
> Track data will be distributed to all display consoles.

In addition, we refine the above consideration in the following manner:

Design Consideration P-6
Track data will be distributed in a multicast manner.

Some of the advantages of this are as follows:

- If a console geometry changes (by panning out or zooming in), the track data will be locally resident on the console.
- If there is a failure of a console, a spare console can be prepared to make a warm start, as opposed to a cold start, since it will already have the track data for its intended sector assignment.

Using multicast to distribute track data is not without a drawback. For example, for each track message, there is the possibility that an interrupt will be raised, which must be handled.

A refinement of the design for distribution of track data is based on the use of message labels. This is stated in the following design consideration:

Design Consideration P-7
Message labels will be used to distinguish the different types of track information (such as new track, update track, etc.).

One of the features of the POSIX.21 interface is the ability to assign a label to a message. The associated label then serves to identify the content of the message. For example, different values of the message label can be used to distinguish a track-update message from a drop-track message.

The design is silent with respect to the possibility of buffering track data messages. For example, one message could be sent that contained several messages. The advantage of this approach would be

- less network bandwidth, since there is only one message header per group of messages as opposed to per message
- less interrupt-level processing (decreased number of interrupts)

One the other hand, some possible disadvantages of buffering messages include

- additional latency

- data dependency: The semantics of a message are data dependent (that is, there would be one message called *Track_Data*, and it would encapsulate new track data, drop track, or update-track data).

The design presented here could be analyzed to assess the impact of track-message buffering. We emphasize, however, that the current design can accommodate either a single-message or buffered-message approach.

5.5.1.3.3 Use of Message Priorities for Conflict-Alert Processing

It is important to be able to process conflict-alert messages with as little delay as possible. We can achieve this in a POSIX.21 design through the use of message priorities. Thus, a message with a high priority will be received (and subsequently processed) sooner than a message with a low priority. A natural case for the use of message priorities is messages that deal with a conflict. Therefore, to address this issue, we have the following design consideration:

> **Design Consideration P-8**
> Messages associated with conflict-alert processing will be sent in a prioritized manner.

The use of message priorities helps to decrease the end-to-end delay associated with a particular message.

5.5.1.3.4 Resulting Design

The design that results from the above discussion is shown in Figure 60.

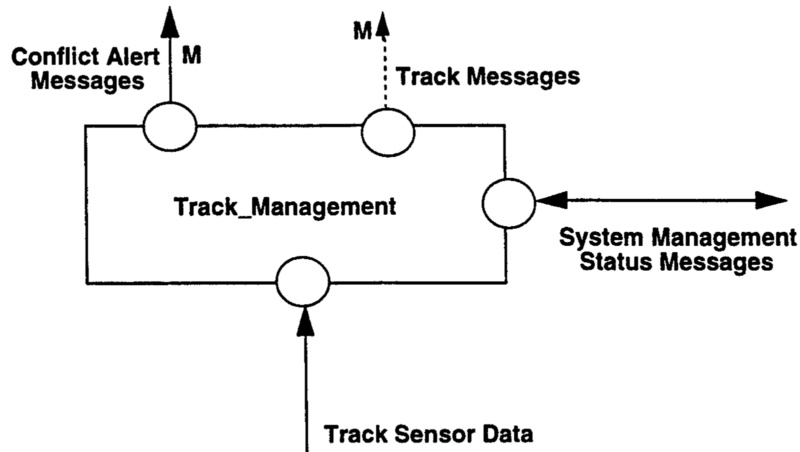


Figure 60: POSIX Design for Track Management

In Figure 60, note that the conflict-alert and track messages are sent in a multicast manner.

5.5.1.4 Flight-Plan Management

5.5.1.4.1 Centralized or Distributed Management

The design for flight-plan management faces similar issues as that of track management. One of these issues concerns how flight-plan data will be managed. One of the main concerns that must be addressed is that flight plans must be correlated with track data. This statement favors the following:

Design Consideration P-9 Flight-plan data will be managed in a centralized manner.

5.5.1.4.2 Distribution of Flight-Plan Data

A second concern deals with the distribution of flight-plan data. One possibility is to distribute a flight plan only to those consoles that require the information. Another option, similar to that for track data, is to multicast flight-plan information. If this is the case, flight-plan information would be available to any console requiring such information. We choose to make the following design consideration:

Design Consideration P-10
Flight-plan messages will be distributed in a multicast manner.

An important consequence of the above design decision is that flight-plan messages will be available to any console that chooses to register for such messages. As in the case of track-message distribution, this design decision will make it easier (and faster) to perform a warm start of a console in the event of a failure.

We must also be able to account for the fact that an operator can perform an action that results in the state change of a flight plan. For example, an operator could change the assigned altitude of a flight plan. We require that flight-plan management maintain a consistent state of all flight plans. In order to accommodate this, we allocate an endpoint that will process state-change messages. The operational semantics of an operator-initiated change would be as follows:

1. An operator changes some state data associated with a flight plan for a flight that is controlled by the operator.[1]
2. The local flight-plan management agent in the console sends a message to flight-plan management indicating the change. The message is expected to be sent in a reliable manner.
3. Flight-plan management then multicasts the modified flight plan to the flight-plan multicast group.

1. We are aware that an operator can make a change to any flight plan through the use of a */OK* override. This is easily accounted for in the present discussion.

The resulting structure for the flight-plan processing appears as shown in Figure 61.

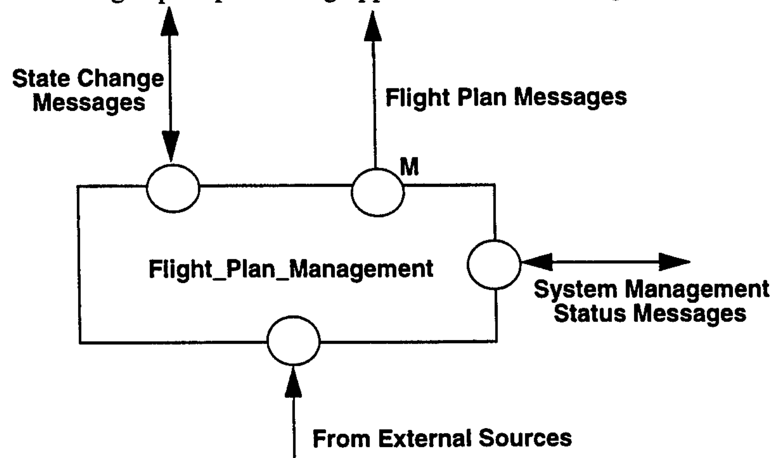


Figure 61: POSIX.21 Design for Flight-Plan Management

5.5.1.5 Displays

The design of the display component is driven by the need to communicate with other components for data management and response to operator commands, including those from system management. The design must therefore account for the following types of communication:

- system management for overall status and management
- airspace management for sector and FPA configuration processing
- track management for the receipt of track messages, including high-priority conflict-alert messages
- flight-plan management for flight-plan data

The overall design is relatively straightforward with the structure shown in Figure 62.

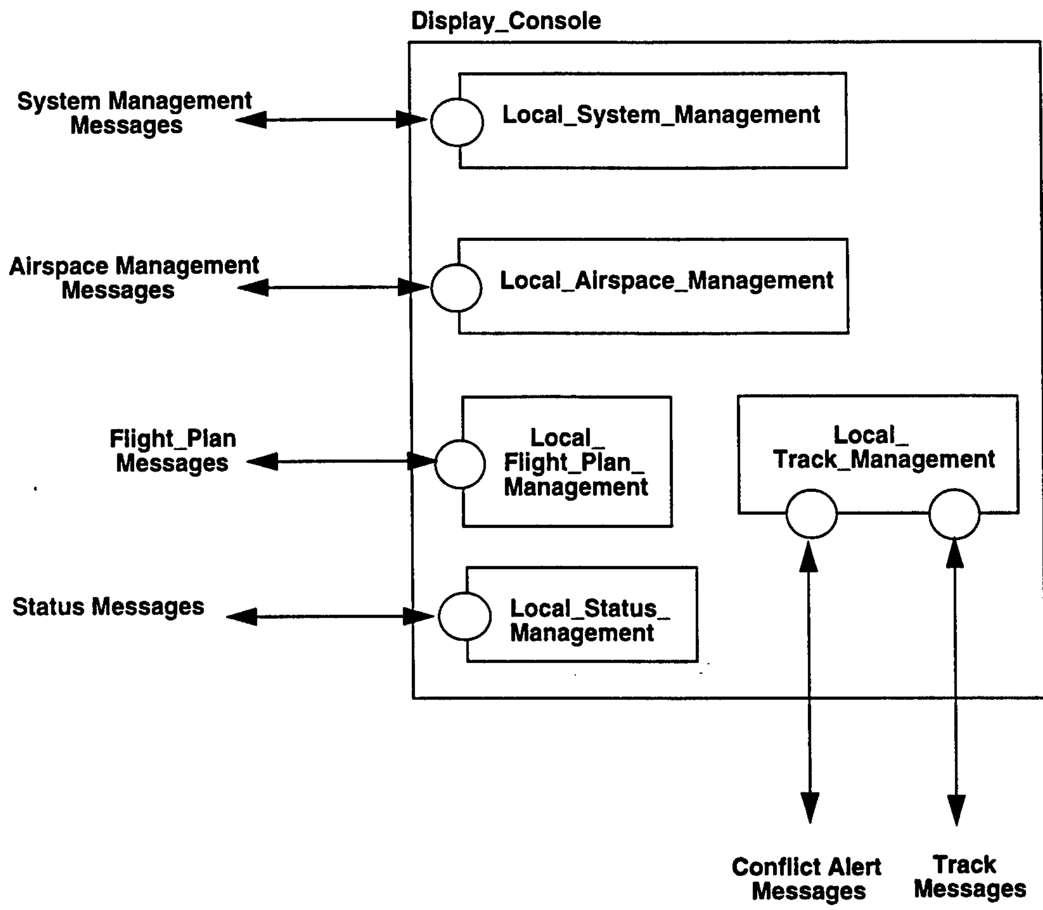


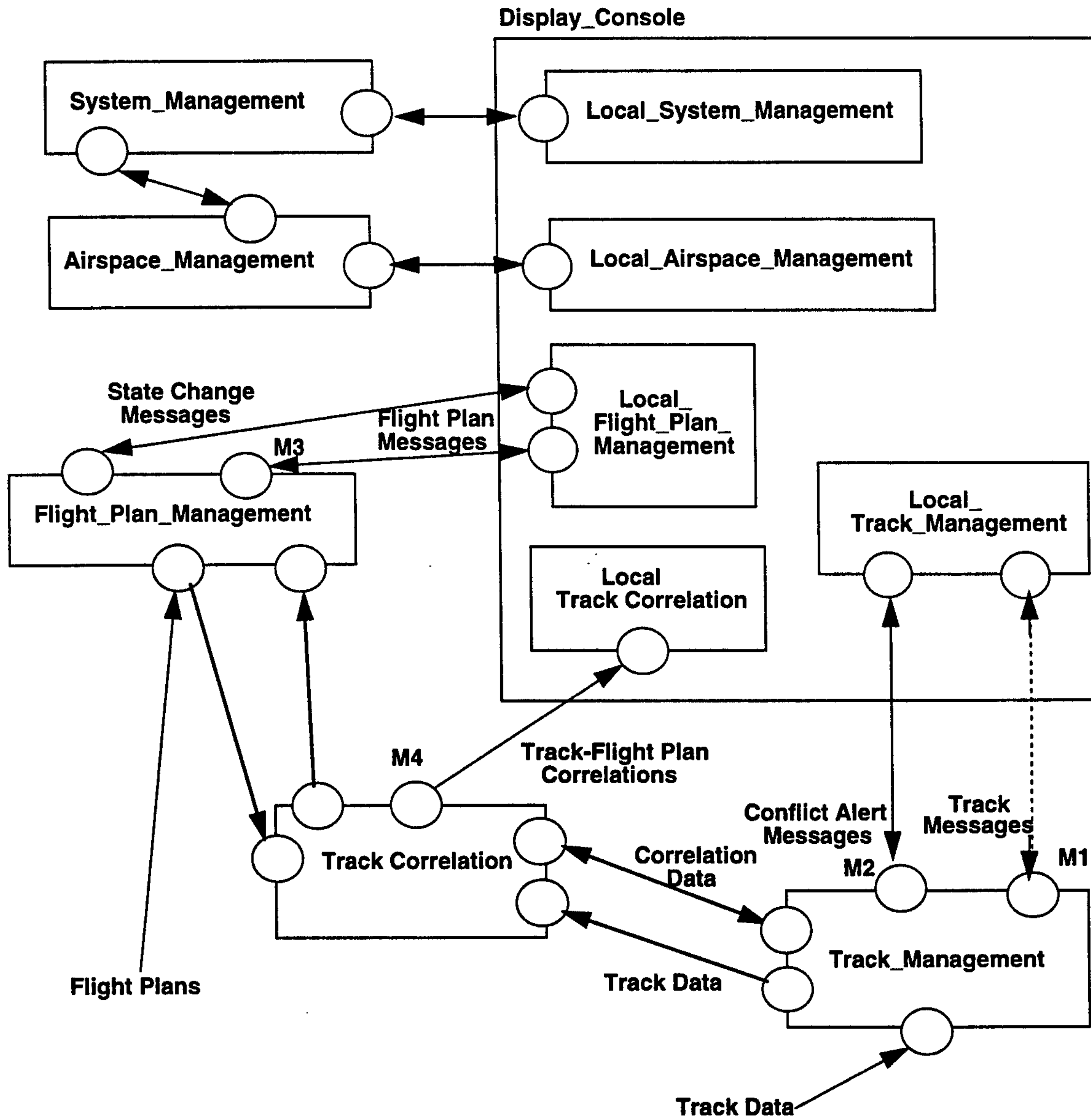
Figure 62: POSIX.21 Design for Display Console

5.5.2 Overall Design

5.5.2.1 Components and Their Interaction

The view that emerges, as illustrated in Figure 63, is that of a collection of replicated, autonomous distributed agents; that is, there is a symmetry between the system functions, such as track management, and the console-specific processing by (symmetric) local agents on that type of data. Another characteristic of the design is that its loose coupling would possibly indi-

cate that it is extensible. For example, if it were necessary to include a time synchronization function, it could be included in Figure 63 in a straightforward manner.



Note: For simplicity, system management status messages are not shown. Mi denotes the "i-th" multicast group.

Figure 63: Overall POSIX.21 Design

5.5.2.2 Additional Comments

5.5.2.2.1 Console-Related Lists

Associated with each console are a number of lists, such as inbound and hold lists. One characteristic of the design is that we assume that list-specific computations are performed at a local console. The alternative would be for the list information to be computed centrally, and then distributed to a particular console (as in the current HCS). For example, each console has information about all flight plans and can therefore determine when a particular flight plan should be added to an inbound list.

5.5.2.2.2 Consistency of State

A major aspect of the POSIX.21 design is that each console maintains global information about all flight plans and tracks. For this approach to be successful, it requires a consistency consideration of state data. This was achieved in the design by using reliable multicast data transfer for track and flight-plan information.

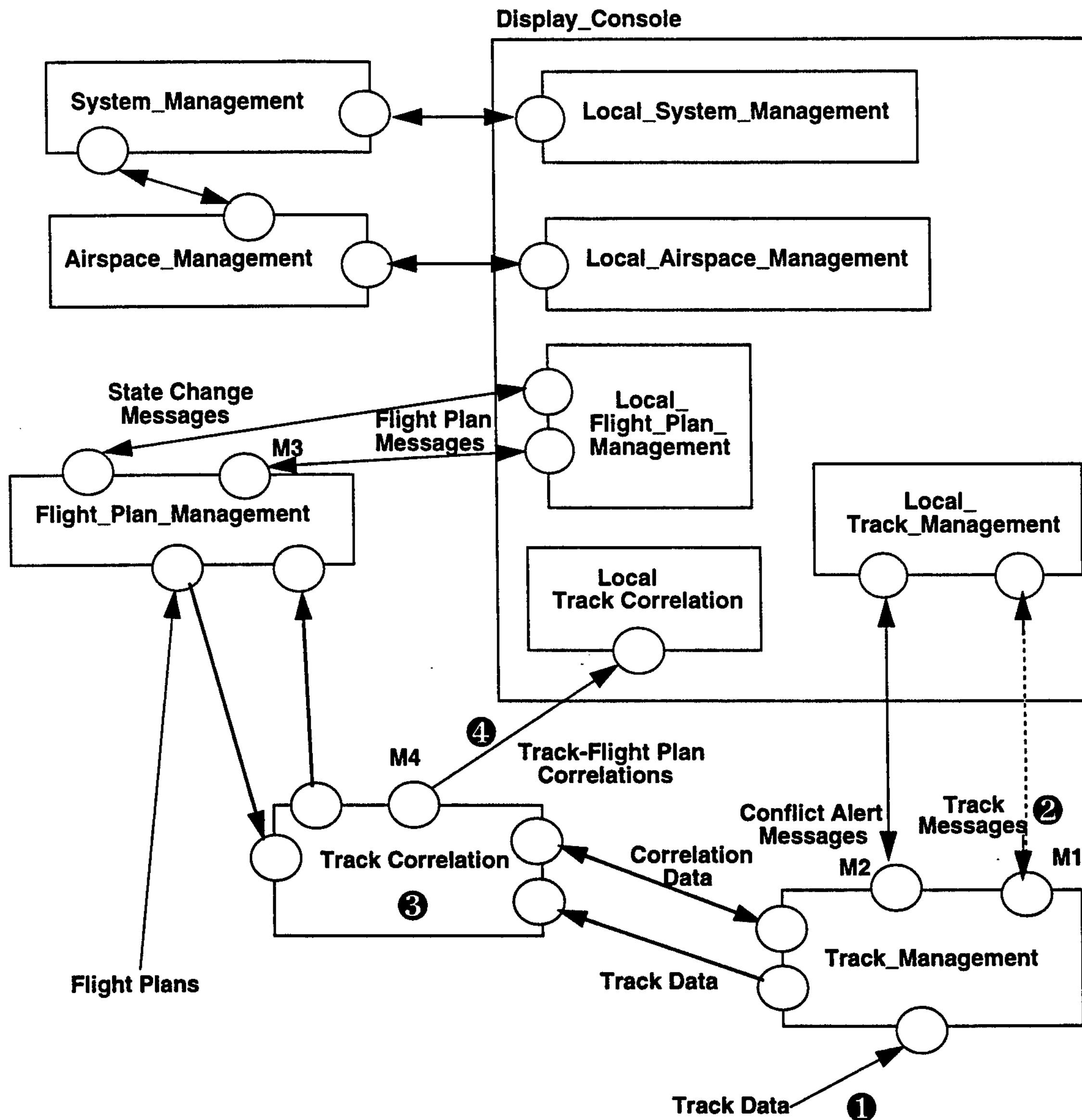
An additional comment, and one that is perhaps more relevant to the existing HCS design, deals with the distribution of state data. We are aware that in the current system, an operator may make a local change to a flight plan, and the associated state data are not propagated to the host, because this would cause flight strips to be printed. This is a procedural workaround to minimize the amount of flight-strip printing. In our design, it is assumed that state data are consistent and therefore distributed to other consoles. We defer the question of flight-strip printing. However, if there is an overriding requirement that each time a flight attribute changes, a flight strip will be printed, a large number of flight strips could be printed.

5.5.2.3 Sample Data-Flow Diagrams

We will again use selected data-flow diagrams to help the reader develop an understanding of the POSIX.21 design. The cases are similar to those used for the CORBA design.

5.5.2.3.1 Track Data

The first case to consider is that in which new track data are available. The distribution of track data is a fundamental operation of an En Route center. A diagram of the data flow for track data is presented in Figure 64.



Note: For simplicity, system management status messages are not shown. Mi denotes the "i-th" multicast group.

Figure 64: Track-Update Data Flow for POSIX.21 Design

The sequence of actions for the distribution of track data is outlined below:

1. New data available: Data containing information about a track is received from the radar. This data will be received by the track-management function, and we assume it is also received by track correlation. This could be accomplished through a multicast message transfer.

2. Distribute track data: Track management multicasts the track data to all consoles. The message is sent in an unreliable manner.
3. Correlate track data: Track correlation processes the new track data to determine if there is a correlation between the received track identification and an existing flight plan.
4. Distribute correlation status: If track correlation determines a new correlation between an existing flight plan and information received for the new track, that information is multicast (reliably) to local track correlation. The correlation information is maintained for each console. For example, when a data block is displayed for a track, the data block also contains the flight plan ID with which the track is correlated.

5.5.2.3.2 Flight-Plan Data

The second important type of data that is processed by an En Route center is that associated with flight plans. The data flow for this case is shown in Figure 65.

The steps required to distribute flight-plan data are as follows:

1. New data available: Flight-plan data are received by flight-plan management, and concurrently, by track correlation. This could be accomplished through a multicast message transfer.
2. Distribute flight-plan data: Flight-plan management multicasts the flight-plan data to all consoles. The message is sent in an unreliable manner.
3. Correlate flight-plan data: Track correlation processes the new flight-plan data to determine if there is a correlation between the received flight plan and an existing track.
4. Distribute correlation status: If track correlation determines a new correlation between an existing flight plan and information received for the new flight plan, the information is multicast (reliably) to local track correlation. The correlation information is maintained for each console. For example, when a data block is displayed for a track, the data block also contains the flight plan ID with which the track is correlated.

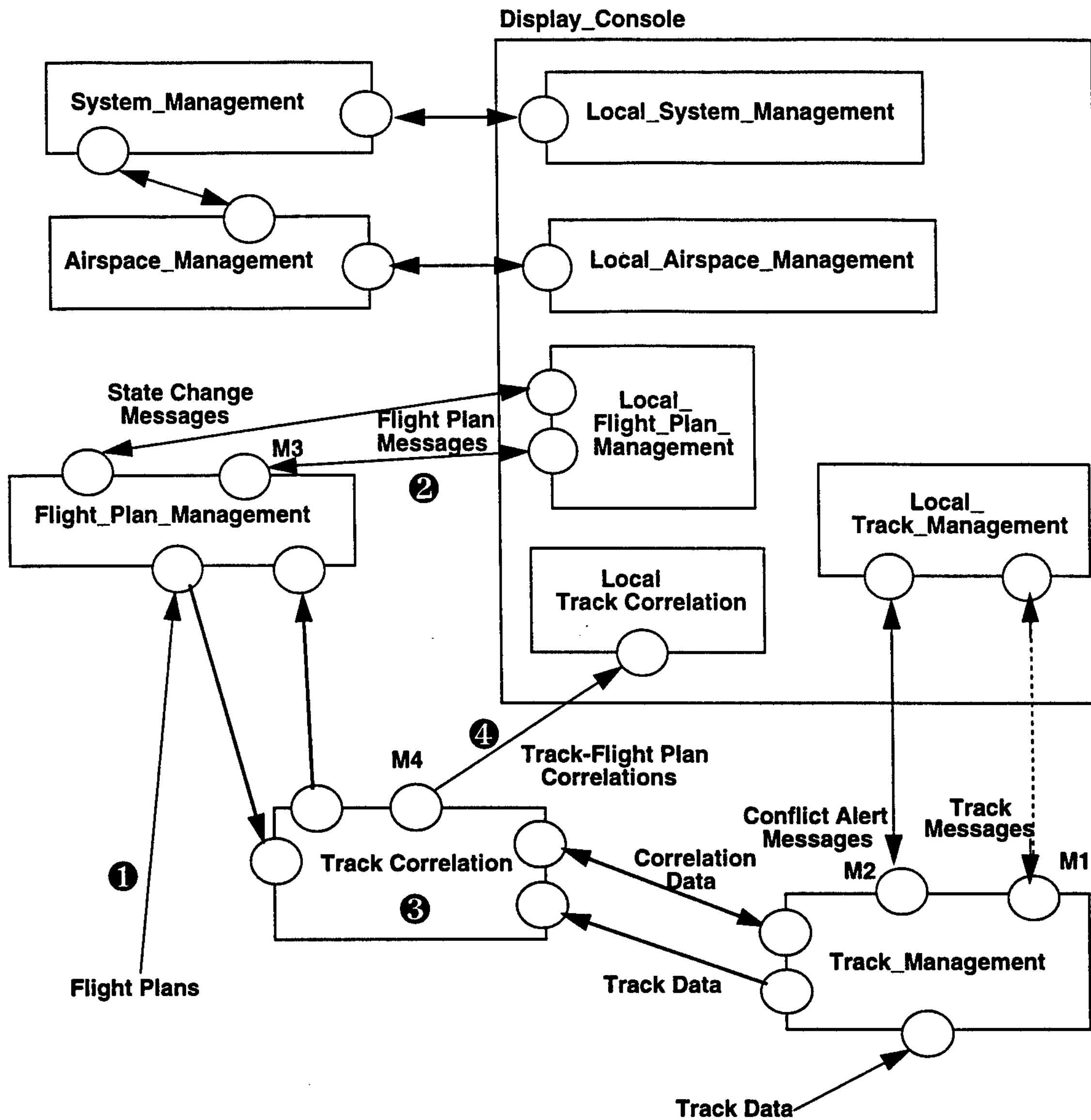


Figure 65: Flight-Plan Data Flow Example

5.5.2.3.3 Sector Combination

One of the main motivations of this work was to deal with the question of sector combination. The data-flow example for this case is presented in Figure 66.

The data-flow diagram illustrates the following steps:

1. Operator request: A system-management operator issues a request to combine two existing sectors. We assume that the request is valid.
2. Update airspace management: System management sends the request for sector combination to airspace management. One reason for this is that one of the functions of airspace management is to maintain the mapping of sectors onto console (physical) addresses. Another reason is that we assume there can be more than one system-management console operating simultaneously; hence, we need a way to serialize and validate sector combination requests. This serialization and validation is assumed to be performed by airspace management.[1]
3. Initiate request of consoles: For the affected consoles (i.e., those that are controlling the airspace with the associated sectors), airspace management sends a message indicating a request to combine the sector. The message is sent in a unicast manner to the local system-management agent. Processing will not be complete until each console intended to be part of the new combination responds in a positive manner. The response indicates a willingness to enter into the combination.
4. Distribute new sector information: Each console must maintain knowledge of other consoles' addresses. This information is needed, for example, for communication between consoles for handoff processing. We assume that the information is multicast and received by the local airspace management agent on each console.

1. Thus, one may think of airspace management as a type of directory service agent.

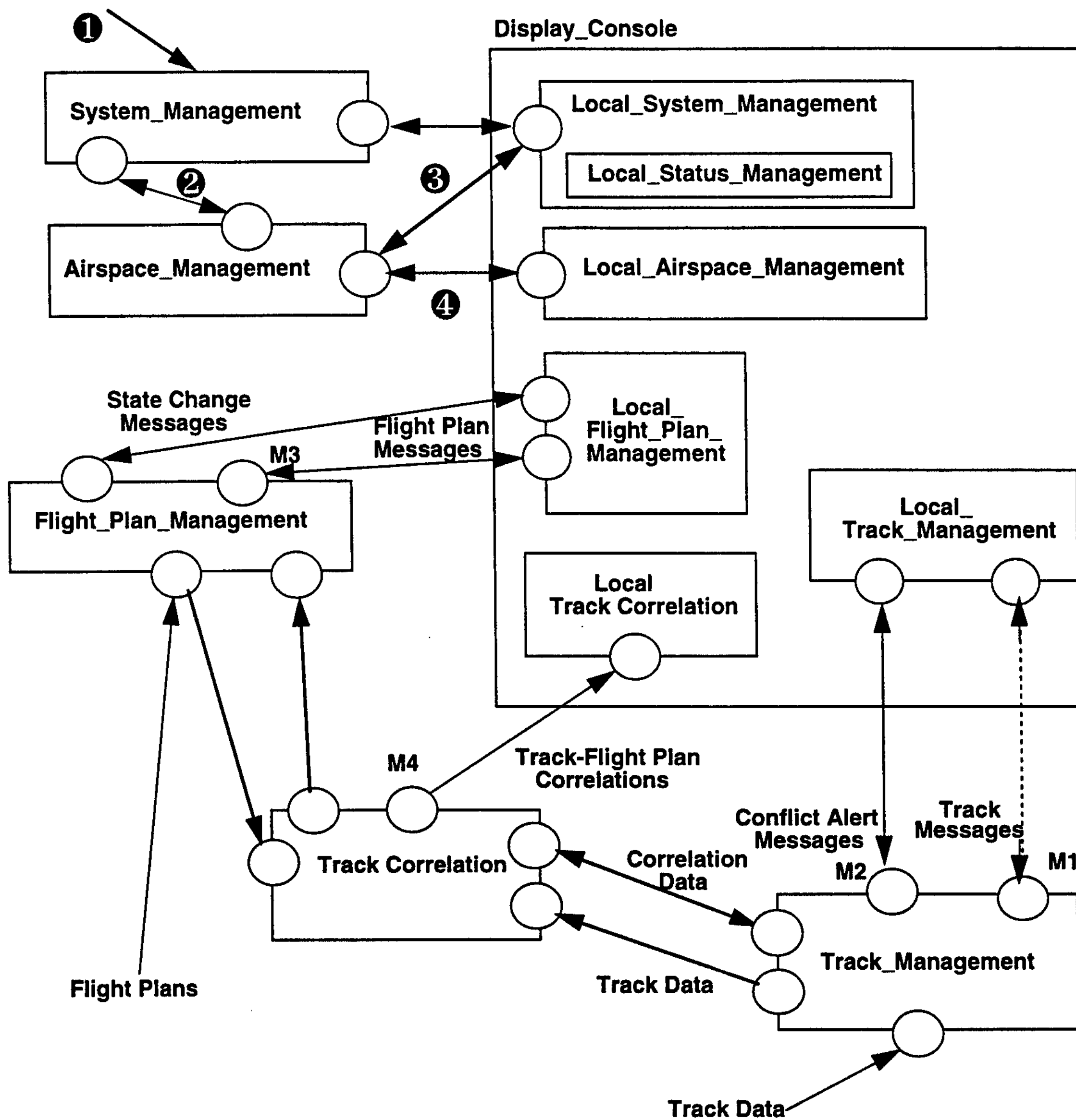


Figure 66: Sector-Combination Data Flow Diagram

Since track and flight plans are distributed in a multicast manner to all consoles, there is no need to reconfigure the new sector with regard to track and flight-plan data. In other words, since each console maintains track and flight-plan state information, a console geometric reconfiguration can proceed in a natural manner.

5.5.2.3.4 Console Failure

There are a number of ways in which a console failure would be detected. For the example data flow, we shall consider the case where the failure is detected by system management. The data-flow diagram for this is shown in Figure 67. We have also included the local status management function in Figure 67 because it is relevant for this example. This is assumed to be a subset of the local system management function.

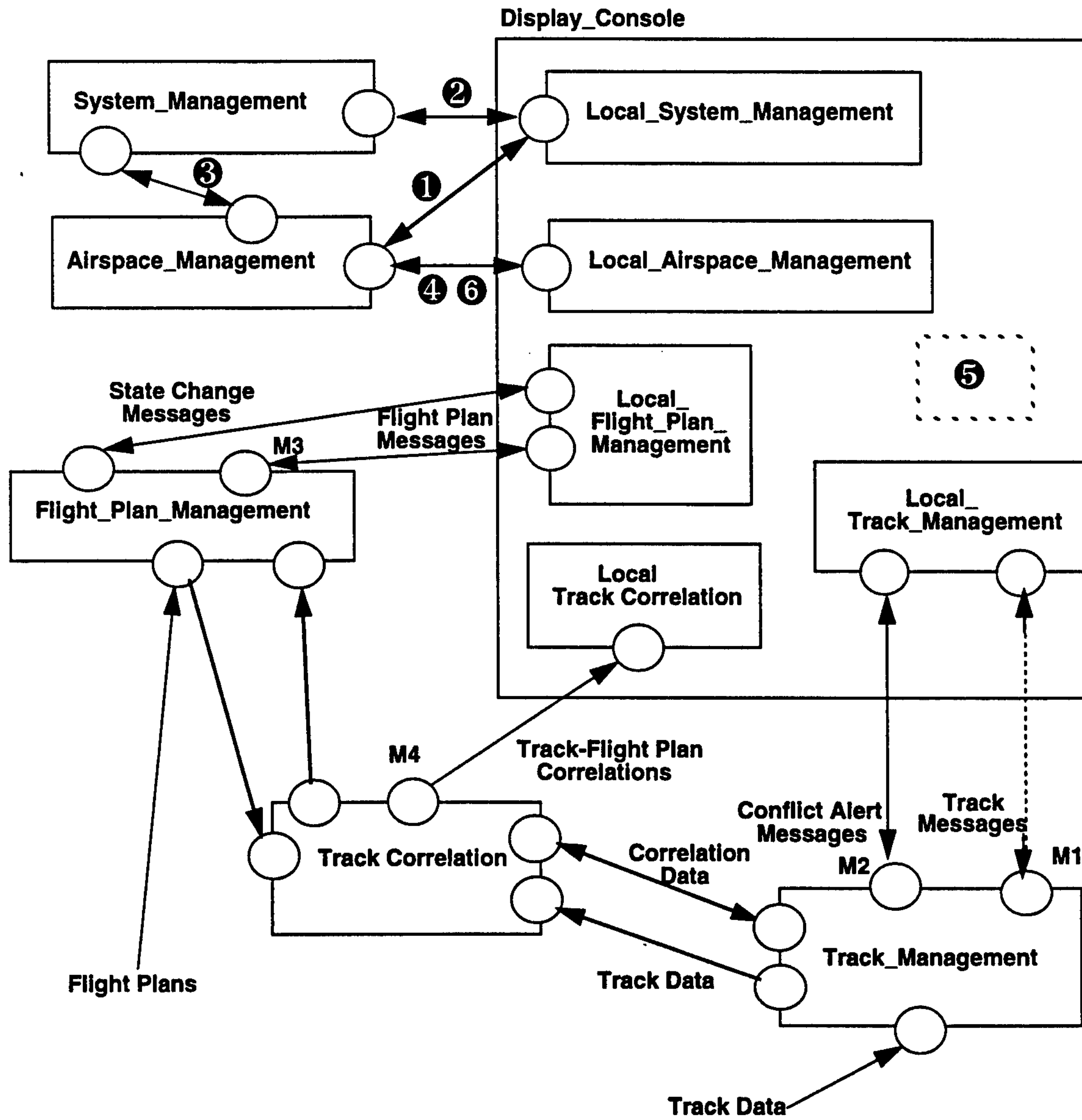


Figure 67: Console-Failure Data Flow Example

The sequence of events for the console failure case is outlined below:

1. Failure of status message: We assume that system management sends a status message at some periodic rate to all devices that it manages in the system. We also assume that this message is sent in a reliable manner (or built on an unreliable request-response model). The message is received by the local status management agent. If there is a console that undergoes a hard failure, the transmission of the status message will fail.
2. Start-up new console: We assume that a spare console exists, which is capable of a warm start. This means that such a console will be receiving track and flight-plan data, but has no assigned sector. When a failure is detected, system management sends a message to a spare console indicating that it should go into operational mode.[1]
3. Initiate airspace-management processing: When the spare console acknowledges the request to become operational, system management informs airspace management.
4. Load console data: Airspace management initiates the transfer of information specific to the sector that is being reconstituted for the spare console. Much of the necessary data is adaptation data (that is, the geometry of the sector, FPAs, airways, console configuration parameters, and so forth). Data are sent to the local airspace management agent on the spare console.
5. Update local state: The last action to make the spare console operational is to determine and/or update local state information, such as what flights were on a hold list. When the local state update is complete, the console is operational.
6. Distribute new sector information: Similar to the discussion of the sector combination data flow, each console must maintain knowledge of other consoles' addresses. This information is needed, for example, for communication between consoles for handoff processing. We assume that the information is multicast and received by the local airspace management agent on each console.

There are two issues associated with console failure and subsequent restart that are worthy of consideration. The first issue deals with the amount of time required to detect the console failure. This is largely associated with the rate at which system status management polls each device. Roughly, the amount of time to detect the failure is equal to the polling period. We would expect that this rate could reasonably be once every few seconds.

The second issue is the time needed to reconstitute a console so that it becomes operational. In terms of the data necessary to achieve this, the two largest data sets involved are the tracks and flight plans. Since track and flight-plan data are being multicast to all consoles, including spare

---

1. To simplify the discussion of Figure 67, we are referring to the same display console object to denote both failed and null consoles.

consoles, there is no need to reconstitute the data. This saves time to transfer all track and flight plans.

Finally, since each console maintains state data for all tracks and flight plans, information about various lists (such as inbound lists) can be computed at a particular console for a given sector. Again, this decreases the overall load by having a given console perform the computation *locally*, rather than in another function such as flight-plan management.

5.5.3 Mapping the Design onto Hardware

As was the case for the CORBA design, different mappings from the software onto hardware are possible. Two likely candidates are

- centralized design: Track management, flight-plan management, and track correlation would reside on the same processor.
- distributed design: Track management, flight-plan management, and track correlation would each reside on a separate processor.

The characteristics of these two designs (which represent ends of a hardware mapping spectrum) are similar to those discussed in Section 4.6.12.2.

5.6 Implementation Concerns

Similar to the discussion in Section 4.7, it is important to understand implementation characteristics of a particular POSIX.21 interface. Some of the items that arose as part of developing the design include the following:

- general performance characteristics: This represents the use of feature benchmarks. For example, what is the time to create or delete an endpoint? Note that this particular case was discussed in Section 5.4 in connection with a design consideration.
- multicast groups: The use of multicast groups provides a powerful design mechanism. However, there are performance considerations that could affect the use of this feature. For example, the POSIX.21 interface semantics could be satisfied through the use of a true multicast data transfer, or through replicated unicast data transfers to group members. Note that the time for these two methods would be quite different. It would be appropriate to know the way in which multicast is implemented.

It is necessary to have performance information for a particular implementation to develop overall system performance models.

5.7 Summary of POSIX.21 Design

The POSIX.21 design includes the following overall characteristics:

- There is a natural mapping of functionality onto components. For example, track management and flight-plan management were easily mapped onto components.
- The design has a lot of symmetry in the context of local components. That is, the symmetry is exploited by having a local track management component in each console that interacts with track management. This allows us to replicate local components on multiple consoles in the architecture.
- Multiple communication models were used as needed. Depending on the requirements, we used unicast, multicast, and labelled-message transfer. The versatility proved extremely useful, allowing us to tailor the communication model to the needs at hand.
- Message priorities were useful. For example, when a track is detected in conflict, a message was sent that had a higher priority than other messages. This eliminated the FIFO queueing of critical messages and would no doubt be useful in other areas as well.

A diagram of the overall design appears in Figure 63 page 133.

6 Assessment of Designs

An architecture is the earliest manifestation of a fledging artifact. The objective of assessing an architecture is to raise one's level of confidence (or fear) that the architecture is heading in the right (or wrong) direction, the philosophy being that major course corrections are easier and cheaper the earlier they are made.

The purpose of this section is to illustrate how architectures can be assessed for various qualities. We illustrate this using performance and modifiability as two important and dissimilar attributes. Performance lends itself to quantitative analysis and draws on mature disciplines such as scheduling theory, while modifiability tends to be more qualitative in nature and does not have such strong theoretical foundations.

Another purpose of this section is to illustrate how to make trade-offs at the architecture level between several quality attributes. This addresses questions such as, "What are the performance ramifications of making a certain class of modifications?"

We do not perform comprehensive analyses for either attribute. Rather, our goal is to highlight the types of questions precipitated by such analyses, the approaches for carrying out the analyses, and the types of conclusions that can be drawn.

6.1 Overview of Principles

6.1.1 Architecture Trade-off Analysis

Quality attributes of large software systems are principally determined by the system's software architecture. That is, in large systems, the achievement of qualities such as performance, availability, and modifiability depends more on the overall software architecture than on code-level practices such as language choice, detailed design, algorithms, data structures, testing, and so forth. This is not to say that the choice of algorithms or data structures is unimportant, but rather that such choices are less crucial to a system's success than its overall software structure, its architecture. Thus, it is in our interest to try to determine, before it is built, whether a system is destined to satisfy its desired qualities.

Although methods for analyzing specific quality attributes exist, these analyses have typically been performed in isolation. In reality however, the attributes of a system *interact*. Performance affects modifiability. Availability affects safety. Security affects performance. Everything affects cost. While experienced designers know that these trade-offs exist, there is no principled method for characterizing them and, in particular, for characterizing the interactions among attributes.

For this reason, software architectures are often designed "in the dark." Trade-off are made—they must be made if the system is to be built—but they are made in an *ad hoc* fashion. Imagine a sound engineer being given a 28-band graphic equalizer, where each of the equalizer's controls has effects that interact with some subset of the other controls. But the engineer is not given a spectrum analyzer and is asked to set up a sound stage for optimal fidelity. Clearly such a task is untenable. The only difference between this analogy and software architecture is that software systems have far more than 28 independent but interacting variables to be "tuned."

There are techniques that designers have used to try to mitigate the risks in choosing an architecture to meet a broad palette of quality attributes. The recent activity in cataloguing design patterns and architectural styles is an example of this. A designer will choose one pattern because it is "good for portability" and another because it is "easily modifiable." But the analysis of patterns does not go any deeper than that. A user of these patterns does not know how portable, modifiable, or robust an architecture is until it has been built.

To address these problems, this report introduces the architecture trade-off analysis method (ATAM). The ATAM is a method for evaluating architecture-level designs that considers multiple quality attributes such as modifiability, performance, reliability, and security in gaining insight into whether the fully fleshed out incarnation of the architecture will meet its requirements. The method identifies trade-off points between these attributes; facilitates communication between stakeholders (such as user, developer, customer, and maintainer) from the perspective of each attribute; clarifies and refines requirements; and provides a framework for an ongoing, concurrent process of system design and analysis.

The ATAM has grown out of work at the Software Engineering Institute on architectural analysis of individual quality attributes: Software Architecture Analysis Method (SAAM) [Kazman 96] for modifiability, performance analysis [Klein 93], availability analysis, and security analysis [Lipson 97]. SAAM has already been successfully used to analyze architectures from a wide variety of domains: software tools, financial management, telephony, multimedia, embedded vehicle control, and so on.

The ATAM, like SAAM, has both social and technical aspects. The technical aspects deal with the kinds of data to be collected and how these data are analyzed. The social aspects deal with the interactions among the system's stakeholders and area-specific experts, allowing them to

communicate using a common framework, to make the implicit assumptions in their analyses explicit, and to provide an objective basis for negotiating the inevitable architecture trade-offs. This report will demonstrate the use of the method and its benefits in clarifying design issues along multiple attribute dimensions, particularly the trade-offs in design.

6.1.2 Why Use Architecture Trade-off Analysis?

All design, in any discipline, involves trade-offs; this is well accepted. What is less well understood is the means for making informed, and even optimal, trade-offs. Design decisions are often made for non-technical reasons: strategic business concerns, meeting the constraints of cost and schedule, using available personnel, and so forth.

Having a structured method helps ensure that the right questions will be asked *early*, during the requirements and design stages when discovered problems can be solved cheaply. It guides users of the method—the stakeholders—to look for conflicts in the requirements and for resolutions to these conflicts in the software architecture.

In doing ATAM analyses, we assume that attribute-specific analyses are *inter*dependent, and that each quality attribute has connections with other attributes, through specific architectural *elements*. An architectural element is a component, a property of the component, or a property of the relationship between components that affects some quality attribute. For example, the priority of a process is an architectural element that could affect performance. The ATAM identifies these dependencies among attributes: what we call *trade-off points*. This is the principal difference between an ATAM analysis and other software analysis techniques—that it explicitly considers the *connections* between multiple attributes and permits principled reasoning about the trade-offs that inevitably result from such connections. Trade-off points arise from architectural elements that are affected by multiple attributes.

6.1.3 The Architecture Trade-off Analysis Method

The ATAM is a spiral model of *design*, as depicted in Figure 68. It is like the standard spiral model in that each iteration takes one to a more complete understanding of the system, reduces risk, and perturbs the design. It is unlike the standard spiral in that no implementation need be involved; each iteration is motivated by the results of the analysis and results in new, more elaborated, more informed designs.

Analyzing an architecture involves manipulating, controlling, and measuring several sets of architectural elements, environmental factors, and architectural constraints.

The primary task of an architect is to lay out an architecture that will lead to system behavior that is as close as possible to the requirements within the cost constraints. For example, performance requirements are stated in terms of latency and/or throughput. However, these attributes depend on the architectural elements pertaining to resource allocation: the policy for allocating processes to processors, scheduling concurrent processes on a single processor, or managing access to shared data stores. The architect must understand the impact of such architectural elements on the ability of the system to meet its requirements and manipulate those elements appropriately.

This task is typically approached with a dearth of tools. The best architects use their hunches, their experience with other systems, and prototyping to guide them. Occasionally, an explicit modeling step is also included as a design activity, or an explicit formal analysis of a single quality attribute is performed.

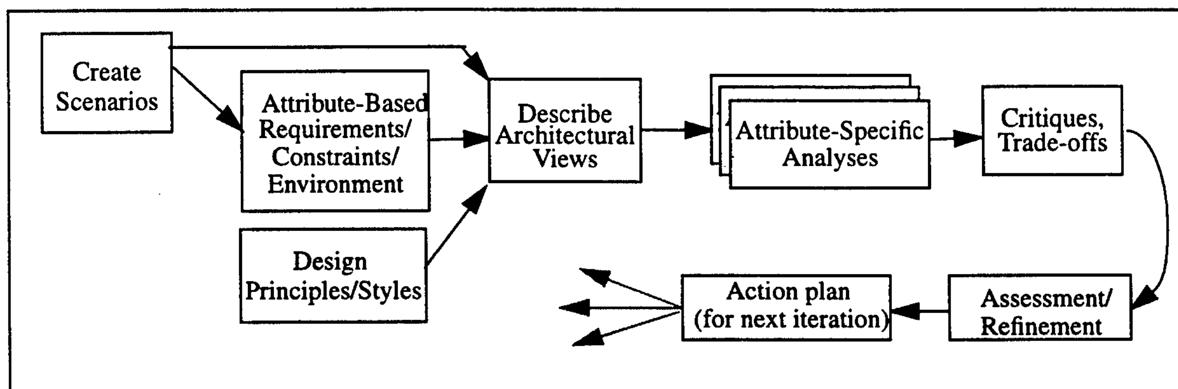


Figure 68: Steps of the Architecture Trade-off Analysis Method

6.1.4 The Steps of the Method

Once a system's initial set of requirements has been elicited and an initial architecture (or small set of architectures) is proposed, subject to environment and other considerations, each quality attribute will be evaluated in turn, and in isolation, with respect to any proposed design. After these evaluations comes a critique step. During this step, trade-off points are found – elements that affect multiple attributes. After the critique, we can either refine the models and re-evaluate; refine the architectures, change the models to reflect these refinements, and reevaluate; or change some requirements.

Step 1 — Create Scenarios

The first step in the method is to elicit system usage scenarios from a representative group of stakeholders. This serves the same purposes as it does in SAAM–to facilitate communication

between stakeholders and to develop a common vision of the important activities that the system should support.

Step 2— Attribute-Based Requirements/Constraints/Environment

The second step in the method is to identify the requirements, constraints, and environment of the system. A requirement can have a specific value or be described via scenarios of hypothetical situations. The environment must be characterized for subsequent analyses (e.g., performance or security), and constraints on the design space, as they evolve, are recorded because they also affect attribute analyses. This step places a strong emphasis on revisiting the scenarios from the previous step to ensure that they account for important quality attributes.

Step 3— Design Principles

Once we believe that we understand our requirements well enough, we can look for engineering principles that help us deal with those requirements. These principles help us to overcome the conflicts inherent in any set of complex requirements. For example, when building a system that needs to be highly available, or highly fault tolerant, an engineering principle to increase the availability of the system is to introduce redundant components. However, these engineering principles must account for variation in the problem domain. For example, if these redundant components always fail by halting, a switch can be inserted that "listens" to the primary component. If this primary component ever stops producing output, the switch detects this and switches to the backup component. On the other hand, if the redundant components can fail "actively"—that is, continue to operate, but produce erroneous output— a more complex switch must be inserted. This more complex switch listens to the (typically three or more) redundant components and then decides, based upon some voting or averaging scheme, which output to use and whether to flag any of the redundant components as failed.

Engineering principles such as "use redundancy to address availability requirements" are general, but can be refined into a particular structural means of addressing the achievement of a system's requirements, based upon the environment in which these principles are applied.

Step 4— Describe Architectural Views

The requirements and design principles together generate candidate architectures and constrain the space of design possibilities. In addition, design almost never starts from a clean slate: legacy systems, interoperability, and the successes/failures of previous projects all constrain the space of architectures. These are described in this step.

Moreover, the candidate architectures are described in terms of the architectural elements that are relevant to each of the important quality attributes. For example, voting schemes are an important element for reliability; concurrency decomposition and process prioritization are important for performance; firewalls and intruder models are important for security; and encapsulation is important for modifiability.

Throughout our description of the method, we assume that *multiple, competing* architectures are being compared. However, designers typically consider themselves to be working on only a *single* architecture at a time. Why are these views not aligned? From our perspective, an architecture is a collection of functionality assigned to a set of structural elements, with constraints on the coordination model—the control flow and data flow among those elements. Almost any change will mutate one of these aspects, thus resulting in a new architecture. While this point might seem like a splitting of hairs, these are important hairs to split in the ATAM context for the following reason: the ATAM requires building and maintaining attribute models (both quantitative and qualitative models) that reflect and help to reason about the architecture. To change any aspect of an architecture—functionality, structural elements, coordination model—will change one or more of the models. Once a change has been proposed, the new and old architectures are "competing," and must be compared: hence, the need for new models that mirror those changes. Using the ATAM, then, is a continual process of choosing among competing architectures, even when these look "pretty much the same" to a casual observer.

Step 5 — Scenario Realization

Scenarios are realized (overlaid) on an architecture to determine their impacts. For example, performance scenarios are run through a system to understand the flow of data and control. Modifiability scenarios are overlaid to understand what components, connections, and interfaces need to be changed.

Step 6 — Attribute-Specific Analyses

Once a system's initial set of requirements and scenarios has been elicited and an initial architecture (or small set of architectures) is proposed, each quality attribute must be analyzed *in isolation*, with respect to each architecture. These analyses can be conducted in any order; no individual critique of attributes against requirements or interaction between attributes is done at this point. Allowing separate (concurrent) analysis is an important separation of concerns that allows individual attribute experts to bring their expertise to bear on the system.

The result of the analyses leads to statements about system behavior with respect to particular attributes: "requests are responded to in 60 ms. average," "the mean time to failure is 2.3

days," "the system is resistant to known attack scripts," "the hardware will cost $80,000 per platform," "the software will require 4 people per year to maintain," and so forth.

Step 7 — Critiques, Trade-offs

The next step of the method is to critique existing models and to find the architectural trade-off points. Although it is standard practice to critique designs, significant additional leverage can be gained by focusing this critique on the *interaction* of attribute-specific analyses, particularly the location of trade-off points. The following paragraph describes how this is done.

First, the sensitivity of individual attribute analyses to particular architectural elements is determined. Once these high-sensitivity architectural elements have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). The availability of that architecture might also vary directly with the number of servers. However, the security of the system might vary inversely with the number of servers (because the system contains more potential points of attack). The number of servers, then, is a trade-off point with respect to this architecture. It is an element, potentially one of many, where architectural trade-offs will be made.

Step 8 — Assessment/Refinement Condition

The purpose of this step is compare the results of the analyses done up to this point to the requirements. When the analyses show that the system's predicted behavior comes adequately close to its requirements, the designers can proceed to a more detailed level of design or to implementation. In practice, however, it is useful to continue to track the architecture with analytic models: to support development, deployment, and beyond to maintenance. Design never ceases in a system's life cycle, and neither should analysis.

Step 9 — Action Plan

In the event that analysis reveals a problem, we now develop an action plan for changing the architecture, models, or requirements. The action plan will draw on the attribute-specific analyses and identification of trade-off points. This step may then lead to another iteration of the method.

In the sections that follow, we will exemplify parts of the ATAM. Space does not permit a complete analysis. The information that we present here is for the purpose of illustrating specific parts of the ATAM.

6.2 Candidate List of Scenarios

Using brainstorming, we came up with the following set of scenarios, some generated with specific attributes in mind (such as performance, modifiability, and reliability) and others with some specific functionality in mind:

- increase in number of flights, for example, 10% / year (modifiability)
- free flight (modifiability)
- dynamic sector boundaries (modifiability)
- introduction of global positioning system (GPS) (modifiability)
- get track data to the display (performance)
- flight plan to display (performance)
- resectorization (performance)
- console failure (reliability)
- multi-center conflict alert (functionality)
- handoffs (functionality)
- starting an idle console (functionality)
- online upgrade (modifiability)

While this list was generated in a relatively short period of time, it represents a potentially rich initial set of scenarios that could/should be applied to host replacement.

6.3 Performance Assessment

6.3.1 Performance Considerations

By performance, we are referring to timeliness, usually measured in terms of the number of events processed in a given interval of time (that is, throughput), or the amount of time required to respond to a specific event (that is, latency). In this section, we consider only latency, and in particular, latency under worst-case conditions.

Performance is basically determined by

- resources
- resource usage
- resource arbitration (including preemptability and queuing)

Resources include processors, networks, memories, and so forth. Resource usage refers to resource consumption in the absence of competition for the resource. Examples are the execution of a process on a dedicated processor or message transmission time on a dedicated network. Resource arbitration refers to the policy for deciding how to allocate a resource to one of several competing requests for the resource. An example is the scheduling policy used for determining which process in the ready queue should execute.

The challenge of performance analysis is to determine if latency requirements will be satisfied given resources, resource-usage patterns, and resource-arbitration policies. The challenge of performance analysis at the architecture level is to derive useful performance predictions in the face of incomplete, inaccurate, or yet-to-be-determined information.

In this report, we confine ourselves to analyzing the latency of processes (and threads) on a collection of processors. Networks and other resources are not explicitly considered. We assume an arbitration policy based on assigning static priorities to processes.

Threads and processes refer to units of concurrency that use the processor. Processes are usually assumed to be "heavier," usually having their own address space, thereby resulting in relatively expensive (when compared with threads) interprocess communication. A process can contain more than one thread. Communication between threads within a process is usually relatively inexpensive. The scheduling of threads is usually performed within the scope of the parent process, but can be between threads of different processes (this is implementation dependent).

The latency of each process/thread is analyzed by examining all of the possible contributors to its latency. Three primary sources of latency are execution time (represented by C in schedulability expressions that appear later), blocking (represented by B), and preemption time, which is accounted for in schedulability equations.

6.3.2 A Performance Scenario

For the purpose of illustrating architecture-level performance analysis, we will consider a single high-level scenario concerned with *resectorization*.[1] This scenario is fleshed out in more detail first using CORBA and then using POSIX.

1. We use the term *resectorization* to refer to operations for combining sectors or combining an FPA with a sector.

Scenario: A system-management operator issues a command requesting resectorization. The resectorization should be completed within the time specified in the system requirements.

A caveat: Clearly, a comprehensive performance analysis cannot be based on only one scenario. This is not our intent. As stated above, our intent is to show how the approach can uncover potential performance problems and offer a modicum of confidence by performing high-level analysis at the architecture level.

In the next two sections, we will take a detailed look at this scenario, first using CORBA (Section 6.3.3) and then using POSIX (Section 6.3.4). At the end of Section 6.3.4, we make several brief remarks about differences between CORBA and POSIX that were highlighted by the analysis.

6.3.3 Performance Using CORBA

In this section, we first look at a realization of the above scenario using the CORBA architecture. This will force us to describe more detail than was in the original object model. Given this additional detail, we then describe a performance model.

The notational conventions shown in Figure 69 are used when augmenting the original designs.

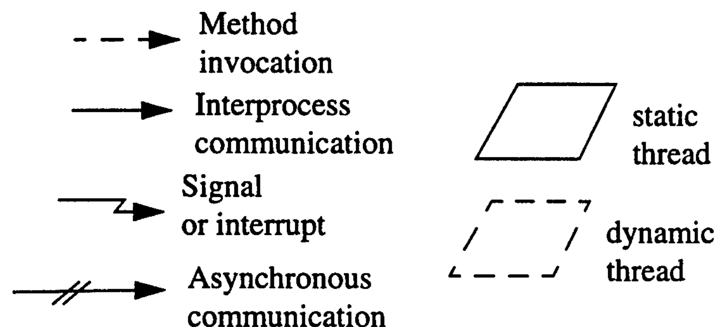


Figure 69: Conventions Used When Augmenting the Original Designs

6.3.3.1 Scenario Realization

A high-level realization of the resectorization scenario is described below:

1. The system management operator initiates a request to combine two (or more) sectors, which results in an *Airspace_Management* method invocation.
2. The *Combine_Sector* method in *Airspace_Management* needs to validate the request (Is the sector number valid, sector state valid, etc.). The method either completes the sector combination request or returns a status indicating why it failed. (That is, the operator does not have to first confirm status and then issue the real request).
3. *Airspace_Management* invokes methods of the *Console_Display_Object*s whose geometry has changed.
4. The active sector (sector into which other sectors are being joined) gets list data from one of the following:
    a. the consoles on which the inactive sectors reside
    b. *Flight_Plan_Management* and *Track_Management*
5. *Airspace_Management* adjusts its view of active sectors.

This high-level realization is discussed in more detail below. It is subdivided into three main parts:

- System management operator initiates a resectorization request
- Combine_Sector method in airspace management is invoked
- Airspace management invokes console display object methods

6.3.3.1.1 System Management Operator Initiates a Resectorization Request

A keyboard (KB) process is responsible for parsing the operator's command and then initiating action that will eventually result in the invocation of the *Combine_Sectors* method of *Airspace_Management*. To ensure that the operator perceives adequate response time, process KB should not be allowed to block for too long and hence should not perform the method invocation itself. Rather, process KB spawns a method invocation thread (MIi) to perform the i'th outstanding method invocation (See Figure 70).

Since CORBA method invocation is synchronous and does not provide a time-out mechanism, we need a mechanism that will return control to thread MIi if a failure occurs that prevents the completion of *Combine_Sectors*. Therefore, thread MIi must start a timer that will generate a signal after a specified amount of time. However, not all implementations of CORBA will necessarily allow a method invocation that is in progress to be interrupted by an OS signal; thus, we need yet another thread, the time-out (TO) thread to accept the time-out signal and termi-

nate thread MIi. When method invocations are timed out or when they complete, a message is displayed on the display console which is managed by the display (DP) thread.

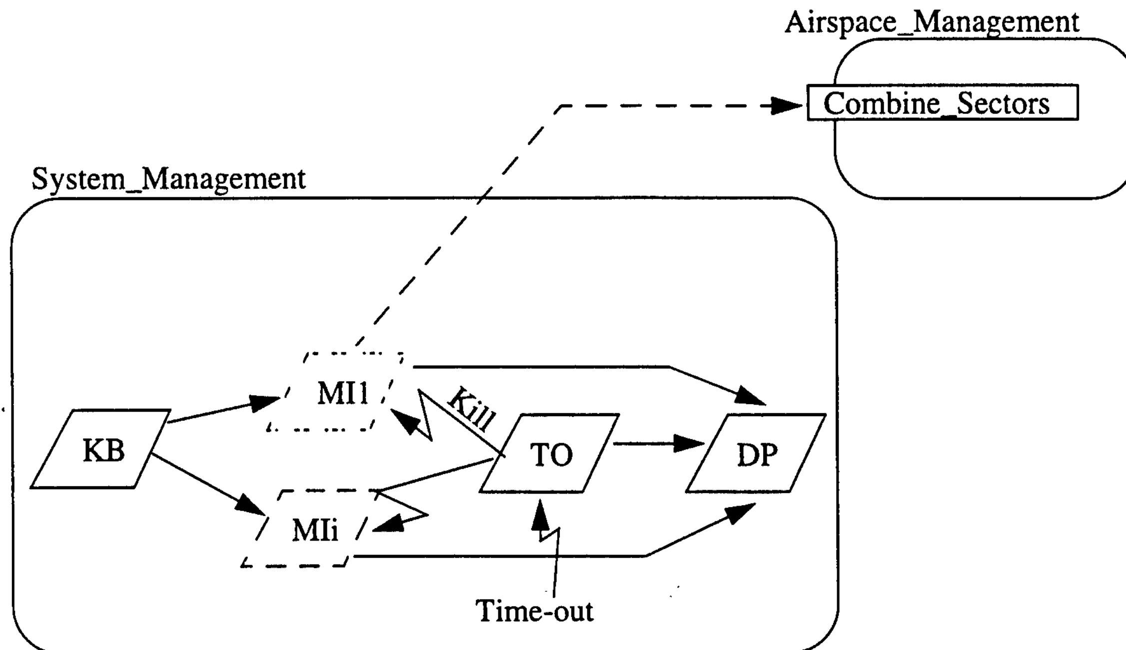


Figure 70: Process/Thread Structure of System_Management

The process of creating a detailed realization of the scenario reveals

- refinements to the scenario and to the requirements
- implicit design constraints and assumptions, which are constraints on the design imposed on the architect/designer
- design considerations, which are constraints placed on the design by the architect/designer carrying the assessment process

6.3.3.1.1.1 Scenario and Requirement Refinements

The following scenario and requirement refinement was noted while constructing the process/ thread structure for *System_Management*:

- Latency for user-interface response time must be bounded.

6.3.3.1.1.2 Design Constraints and Assumptions

Design constraints and design assumptions that became evident as a consequence of realizing this scenario using CORBA follow:

- Assume that thread creation is relatively inexpensive.
- Assume that CORBA method invocation might not be interruptible by operating system (OS) signals.
- *System_Management* and *Airspace_Management* objects reside on different processors as shown in Figure 70.
- Assume that the CORBA server is multi-threaded (to accommodate several outstanding method invocations to the same CORBA object).
- Assume that signals can have data passed with them so that TO knows which method invocation has been timed out
- Assume that signals are reliably queued.

6.3.3.1.1.3 Design Considerations

The following design considerations have an impact on the subsequent performance analysis:

- Based on the assumption of inexpensive thread creation, *System_Management* has one static process that spawns threads for carrying out the appropriate method invocation.
- Priority-based preemptive scheduling is used to schedule processes/threads. (Otherwise, the scheduling of threads in a multi-threaded environment is not defined and might result in, for example, round-robin scheduling, resulting in less than the desired amount of control and hence a loss of predictability.)
- Assume that the processes/threads of *System_Management* have the following priorities: KB - 10; MIs - 20; TO - 30; DP - 15. (A larger number means a higher priority.)
- A mechanism is needed to time out a method invocation that appears (has taken more than a tolerable amount of time) to have failed. Reliable time-out signals are being used for this purpose.

6.3.3.1.2 Combine_Sector Method in Airspace Management Is Invoked

The *Combine_Sectors* method is responsible for validating the command (e.g., ensuring proper sector status and correct sector numbers), for figuring out which consoles are responsible for which sectors, and invoking methods on the appropriate consoles. *Airspace_Management* is also responsible for serializing overlapping resectorization requests. A process architecture that is similar to the one used for *System_Management* is also appropriate here and is shown in Figure 71.

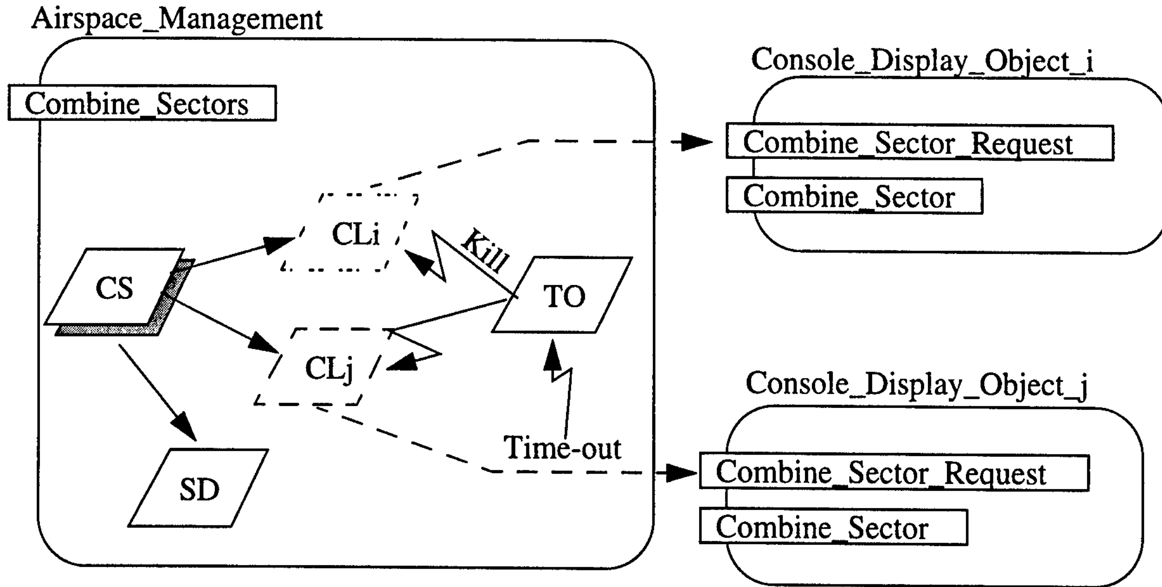


Figure 71: Process/Thread Structure of Airspace_Management

Notice that we are assuming that methods are "reentrant," that is, a single method can have more than one active invocation. This is represented by the shaded task in the above figure. This allows multiple resectorization requests involving disjoint sets of sectors to proceed concurrently. Since the ORB needs to spawn a process for this purpose, we explicitly represent the ORB as shown below in Figure 72.

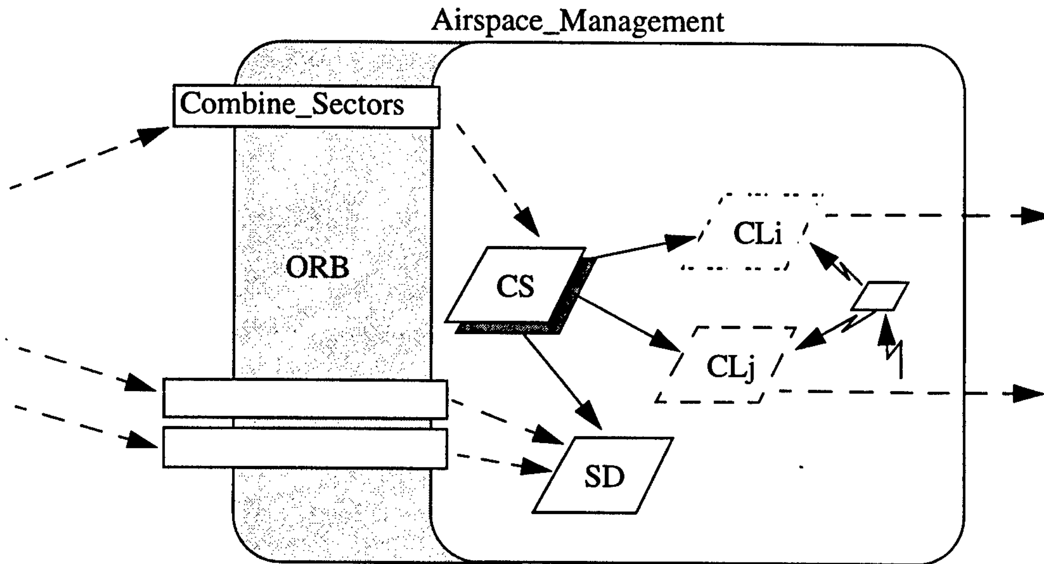


Figure 72: ORB and Airspace_Management

Sector data (SD) must be examined to determine which consoles are affected. For example, *Airspace_Management* maintains the state of a sector including the assigned console and whether it is active, inactive, or in transition. For each affected console, a thread is spawned (CLi) to invoke the *Combine_Sector_Request* and *Combine_Sector* methods for each appropriate console. The same time-out mechanism used in *System_Management* is used here. Since sector data are shared by multiple methods, mutual exclusive access must be enforced. This is handled by thread SD.

Airspace_Management invokes the *Combine_Sector_Request* method on the new controlling console and on one or more consoles being combined and requesting permission to commence resectorization. If all consoles respond in the affirmative, *Airspace_Management* changes its sector data so that flight plan and track data are sent to all consoles involved in the resectorization, and *Airspace_Management* directs all of the involved consoles to commence resectorization by invoking the *Combine_Sector* method on each involved console. When all consoles indicate that resectorization has been completed, *Airspace_Management* lets all of the consoles know that the new sector assignments can be put into effect.

Airspace_Management also performs a data-routing function. It routes flight-plan-related and track data from *Flight_Plan_Management* and *Track_Management*, respectively, to the appropriate consoles. This can result in contention for sector data. We will assume that new flight-plan-related data arrive once per T_{FP} seconds on the average, and new track information arrives once per T_{TK} seconds for each of $N_{Flights}$.

6.3.3.1.2.1 Scenario and Requirement Refinements

The following scenario and requirement refinements were noted while constructing the process/thread structure for *Airspace_Management*:

- *Airspace_Management* is responsible for the "serialization" of requests to combine sectors. For example, if a request to combine sectors 1 and 2 is underway, and a request for combining sectors 2 and 3 is received, *Airspace_Management* must wait until the first request has been processed before starting the second. Moreover, it might be prudent to send status information to the system management operator reflecting this type of situation.
- Assume that new flight-plan-related data arrives once per T_{FP} seconds.
- Assume that a track update arrives once per T_{TK} seconds for each of $N_{Flights}$.

6.3.3.1.2.2 Design Constraints and Assumptions

Design constraints and assumptions that became evident as a consequence of realizing this scenario using CORBA follow:

- Assume predictable termination semantics when CLi is "killed" by TO; that is, CS receives control within a relatively short amount of time after CLi is killed and also knows that the return occurred under exceptional conditions.
- Since the ORB manages the routing of a method invocation from source to destination and there can be many concurrent invocations, assumptions must be made about how contention for the ORB is handled:
  - Assume that the ORB queue is unlimited, and therefore no requests are "turned away" for later retry.
  - Assume that the ORB queue requests are handled in a FIFO manner.

6.3.3.1.2.3 Design Considerations

The following design considerations have an impact on the subsequent performance analysis:

- SD requires a locking mechanism to manage contention for shared data (such as sector state). Assume that there are multiple concurrent readers, but only a single writer is allowed.
- Assume that SD has been assigned a priority that is higher than the priority of any of its clients (thereby achieving a blocked-at-most-once property[1]).

6.3.3.1.3 Airspace Management Invokes Console Display Object Methods

Combine_Sector_Request and *Combine_Sector* are invoked by *Airspace_Management* on each affected console object (See Figure 73). *Combine_Sector_Request* queries console operators to determine whether a change in sector geometry is viable. If the operator consents (and all other criteria are met), an affirmative indication is sent back to *Airspace_Management*. If all consoles consent, *Airspace_Management* requests that resectorization commences by invoking *Combine_Sector*. The controlling console gathers data from *Flight_Plan_Management* and *Track_Management* to regenerate list data (i.e., the hold, conflict-alert and inbound lists). If it receives list data within a certain time interval, it then sends a completion indication to *Airspace_Management*. When *Airspace_Management* lets all consoles know that resectorization has been completed, the active console assumes control, and the inactive console relinquishes control. This protocol is handled by the console management (CM) process.

---

1. A high-priority thread benefits from a blocked-at-most-once property when its worst-case blocking time is derived from a single critical section of a lower priority thread. On a uniprocessor, when multiple threads access a shared resource (such as sector data) and the priority of the critical section is higher than the highest priority thread which accesses the data and execution is not suspended during the critical section, a block-at-most-once property is achieved.

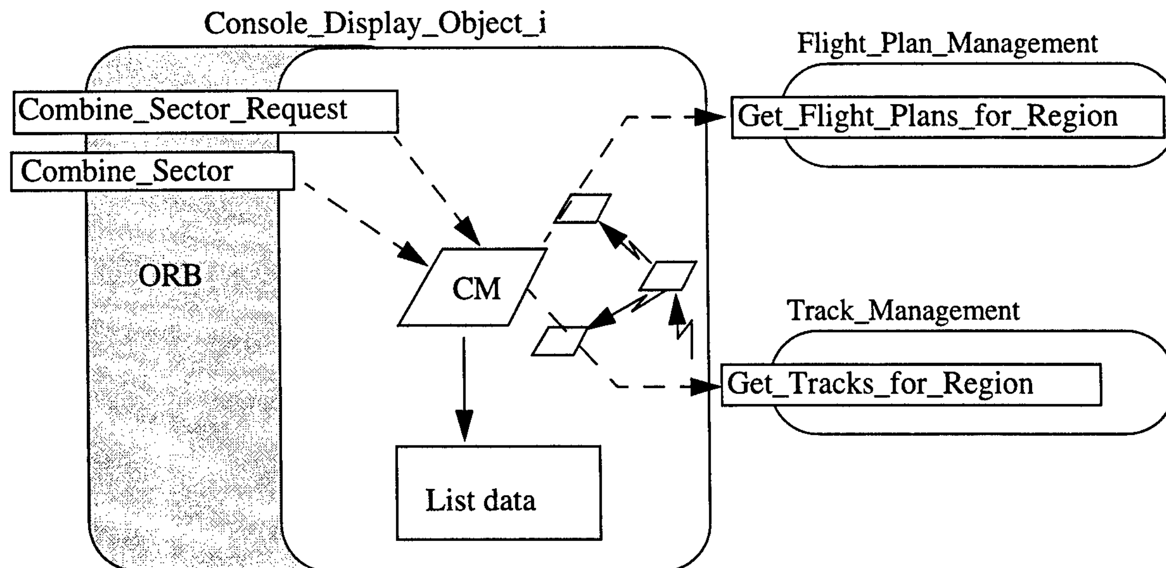


Figure 73: Process/Thread Structure of Console_Display_Object

Then, the console must make any appropriate geometry changes and communicate those changes back to *Airspace_Management.*

6.3.3.1.3.1 Scenario and Requirement Refinements

The following scenario and requirement refinements were noted while constructing the process/thread structure for the *Console_Display_Object*:

- There will be a short window of time in which two consoles can receive data for a sector. This window occurs after all consoles agree to participate in resectorization and before the consoles are notified that resectorization has been completed. A maximum duration for this "short" window must be specified.

6.3.3.1.3.2 Design Constraints and Assumptions

No additional design constraints or assumptions were revealed when examining the *Console_Display_Object.*

6.3.3.1.3.3 Design Considerations

After a sector has been added or deleted from the purview of a console, its list data must be acquired by the active console. There are two options for how to approach this:

- Invoke methods on *Flight_Plan_Management* and *Track_Management* to acquire data.
- The active console requests the data from the inactive console(s).

The first option was chosen because it ensures the consistency of state data. The second option has potential performance advantages if the consistency of state data can be ensured.

6.3.3.2 Performance Modeling

Using the scenario discussed in the previous section, we develop performance models in this section. Again, be aware that the performance models developed in this section are representative of what can be done, but are by no means comprehensive. We have not considered all of the preemptive and blocking effects, nor have we modeled resources other than the processor; most notably we have not modeled the network. Nevertheless, a comprehensive model requires the same type of reasoning used here and undoubtedly will provide more insights than we have gained through a cursory analysis.

Once again, we consider the scenario in three stages:

- System management operator initiates a resectorization request
- Combine_Sector method in airspace management is invoked
- Airspace management invokes console display object methods

After the three stages are discussed, end-to-end latency for the scenario is discussed.

6.3.3.2.1 System Management Operator Initiates a Resectorization Request

Since the operator response time must be bounded, in this section we consider the latency associated with KB. It receives a command (which has already been echoed to the display) and must parse it and spawn a thread to carry out the resulting method invocation. The computation time for this is

$$C_{KB} + C_{MI\text{-}start} \tag{6-1}$$

MI-start includes thread creation, the starting of the timer, data marshalling, and the method invocation–all of which occur in thread MIi at a priority 20. Note that thread KB cannot continue to execute until MIi is suspended since MIi's priority is higher than KB's priority.

To emphasize the various aspects of MI-start an alternate, more detailed expression could be used:[1]

$$C_{KB} + C_{Thread\text{-}create} + C_{Timer\text{-}start} + C_{Marshall} + C_{Invocation} \quad (6\text{-}2)$$

There is no blocking of KB due to the sharing of resources with other threads and hence no "B terms" in the above expression. However, since KB has the lowest priority, there will be occasions in which KB and "MI-start" are preempted by either the return or the timing out of the method invocations. Assuming n_{inv} outstanding method invocations and that in the worst case they all preempt either KB or MI-start, the expression accounting for preemption is

$$n_{inv} \left[\max(C_{TO}, C_{MI\text{-}end}) + C_{DP}\right] \quad (6\text{-}3)$$

The value of TO includes the time to field the time-out signal and kill thread MI. MI-end includes unmarshalling time, and time to process status information from the method invocation, send this information to DP, and then terminate. The max function is needed since either the time-out or the return from method invocation occurs, not both.

The worst-case latency for KB is the sum of the first (or second) expression and the third expression:

$$C_{KB} + C_{MI\text{-}start} + n_{inv} \left[\max(C_{TO}, C_{MI\text{-}end}) + C_{DP}\right] \quad (6\text{-}4)$$

1. In general, we will not use this level of detail in the schedulability expressions. Rather, we will use less detail and tables to explain what aspects of computation are denoted by each term.

The schedulability model for determining KB's worst-case latency is shown in Table 6.

Process / Thread	Priority	Description of Computation	Schedulability Expression
KB	10	KB computation	$C_{KB} + C_{MI\text{-}start}$
MI-start	20	Create thread Start timer Marshall data Invoke method	
TO	20	Field time-out signal Kill thread	$\max(C_{TO}, C_{MI\text{-}end}) + C_{DP}$
MI-end	20	Return from method invocation Unmarshall data Process status information Send to DP Terminate	
DP	15	Display status	
"KB latency" = $C_{KB} + C_{MI\text{-}start} + n_{inv}\,[\max(C_{TO}, C_{MI\text{-}end}) + C_{DP}]$			

Table 6: Schedulability Model for Keyboard Latency Using CORBA

Note that the above analysis serves two purposes: (1) to describe the latency for KB, and (2) to get insight into the contribution of *System_Management* to the end-to-end latency for resectorization.

6.3.3.2.2 Combine_Sector Method in Airspace Management Is Invoked

Since we assume that the ORB uses a FIFO queue for outstanding method invocations, we assume that in the worst case, *System_Management* invokes *Combine_Sectors* immediately after all $N_{Flights}$ track-update requests (through the *Track_Notify* method) have been made.[1] The blocking time experienced is

$$N_{Flights} B_{ORB} \quad (6\text{-}5)$$

1. From the specifications of loading discussed in Appendix D, it is evident that track updates dominate updates associated with flight plans. For this analysis, we ignore the effects of flight-plan-related updates.

Eventually, the ORB activates CS, which does some computation after data is unmarshalled[1] and then queries SD to find out which consoles are involved in resectorization. Since it is possible that the needed data are currently being changed by some other thread, CS might be blocked before reading the database. These three additional terms for the schedulability expression are:[2]

$$C_{CS} + B_{SD\text{-}write} + C_{SD\text{-}read} \tag{6-6}$$

Each CL thread takes time to be created, set its timer, marshall data, and commence method invocation. As for *System_Management*, there is work to be performed upon the return or time-out of the method invocation. These two additional terms are

$$n_{combine}\,[C_{CL\text{-}start} + \max(C_{TO}, C_{CL\text{-}end})] \tag{6-7}$$

Finally, the remote execution for each console needs to complete before being timed out. Assuming that it does not time out, the expression follows (where L_{CLi} stands for the latency associated with each console[3]):

$$\max(L_{CLi}, L_{CLj}) \tag{6-8}$$

The result of adding together expressions 6-5, 6-6, 6-7, and 6-8 is:

$$N_{Flights}B_{ORB} + C_{CS} + B_{SD\text{-}write} + C_{SD\text{-}read} + n_{combine}\,[C_{CL\text{-}start} + \max(C_{TO}, C_{CL\text{-}end})] + \max(L_{CLi}, L_{CLj}) \tag{6-9}$$

Note that we are making several simplifying assumptions. It is likely that there are other computations being performed by *Airspace_Management* other than those for resectorization. These other computations could contribute additional latency through preemptive effects (of course, depending on the priority at which they are carried out). Other scenarios would highlight these computations. The performance models should then be incrementally updated to reflect these new computations.

---

1. While unmarshalling is handled by the ORB, it is associated with CS for performance-modeling purposes.

2. Note that the priority of SD plays an important role in determining the blocking time. It has been assigned a "ceiling" priority that is higher than the priority of any client threads that call it. In doing so, we limit how long a high-priority client has to wait to the longest critical of all lower priority clients.

3. Strictly speaking, the latency term is actually several latency terms. *Airspace_Management* and *Console_Display_Object* engage in a protocol involving several method invocations. We model this as a single method invocation.

Also note that we are modeling the case in which there is only a single concurrent resectorization. If more than one resectorization is proceeding simultaneously, one could preempt the other and add to its latency.

Table 7 summarizes the performance model for *Airspace_Management*.

Process / Thread	Priority	Description of Computation	Schedulability Expression
n/a	n/a	Queuing in the ORB Unmarshalling track data	$N_{Flights}B_{ORB}$
CS	20	Unmarshall data Possibly block on SD	$C_{CS} + B_{SD\text{-}write} + C_{SD\text{-}read}$
SD	35	Read or write sector data	
CL-start	30	Create thread Start timer Marshall data Invoke method	$C_{CL\text{-}start} + \max(C_{TO}, C_{CL\text{-}end})$
TO	30	Field time-out signal Kill thread	
CL-end	30	Return from method invocation Unmarshall data	
n/a	n/a	Latency for console operations	$\max(L_{CLi}, L_{CLj})$

"Combine Sectors Latency" =

$$N_{Flights}B_{ORB} + C_{CS} + B_{SD\text{-}write} + C_{SD\text{-}read} + n_{combine}\,[C_{CI\text{-}start} + \max(C_{TO}, C_{CI\text{-}end})] + \max(L_{CIi}, L_{CLj})$$

Table 7: Schedulability Model for Combine_Sectors in Airspace_Management

6.3.3.2.3 Airspace Management Invokes Console Display Object Methods

Assume that there are about $n_{consoles}$ consoles that evenly control $N_{Flights}$ tracks.

Combine_Sector_Request and *Combine_Sector* are invoked by *Airspace_Management* on each affected console object. This causes the string of execution to "fork" when the different methods are invoked and "join" when all of the method invocations complete. We will focus

on the *active console's* branch since it is likely to be the one that takes the longest amount of time to complete. That is, it is likely to be the value of $\max(L_{CLi}, L_{CLj})$.

The ORB is a potential bottleneck here as it was for *Airspace_Management*. Assuming that either the *Combine_Sector_Request* or *Combine_Sector* method is invoked immediately after a batch of $N_{Flights}/n_{consoles}$ tracks arrive at the console, there is a blocking term:

$$(N_{Flights}/n_{consoles})B_{ORB} \quad (6\text{-}10)$$

After data are unmarshalled, the next step is for CMi to perform its computation and then gather data from *Flight_Plan_Management* (FPM) and *Track_Management* (TM) for reconstructing list data. Each of these databases is a potential source of blocking:

$$C_{CMi} + C_{TM} + C_{FP\,M} + B_{TM} + B_{FPM} \quad (6\text{-}11)$$

Finally, the possibility exists that list data are being accessed by another thread causing further blocking:

$$C_{LD} + B_{LD} \quad (6\text{-}12)$$

We assume that any changes in console geometry are communicated back to *Airspace_Management* after resectorization is completed.

Table 8 summarizes the performance model for *Console_Display_Object*:

Process / Thread	Priority	Description of Computation	Schedulability Expression
ORB	n/a	Blocking due to ORB Unmarshalling track data	$(N_{Flights}/n_{consoles})B_{ORB}$
TM	n/a	Blocking due to TM	$C_{CMi} + C_{TM} + C_{FP\,M} + B_{TM} + B_{FPM}$
FPM	n/a	Blocking due to FPM	
CMi	20	Local computation Get flight plan data Unmarshall Get track data Unmarshall	
CMi	20	Blocking due to LD Process list data	$C_{LD} + B_{LD}$
"Assign_Sector Latency" = $(N_{Flights}/n_{consoles})B_{ORB} + C_{CMi} + C_{TM} + C_{FP\,M} + B_{TM} + B_{FPM} + C_{LD} + B_{LD}$			

Table 8: Schedulability Model for Console_Display_Object

6.3.3.2.4 End-to-End Latency for Resectorization

Figure 74 summarizes the resectorization string of computation.

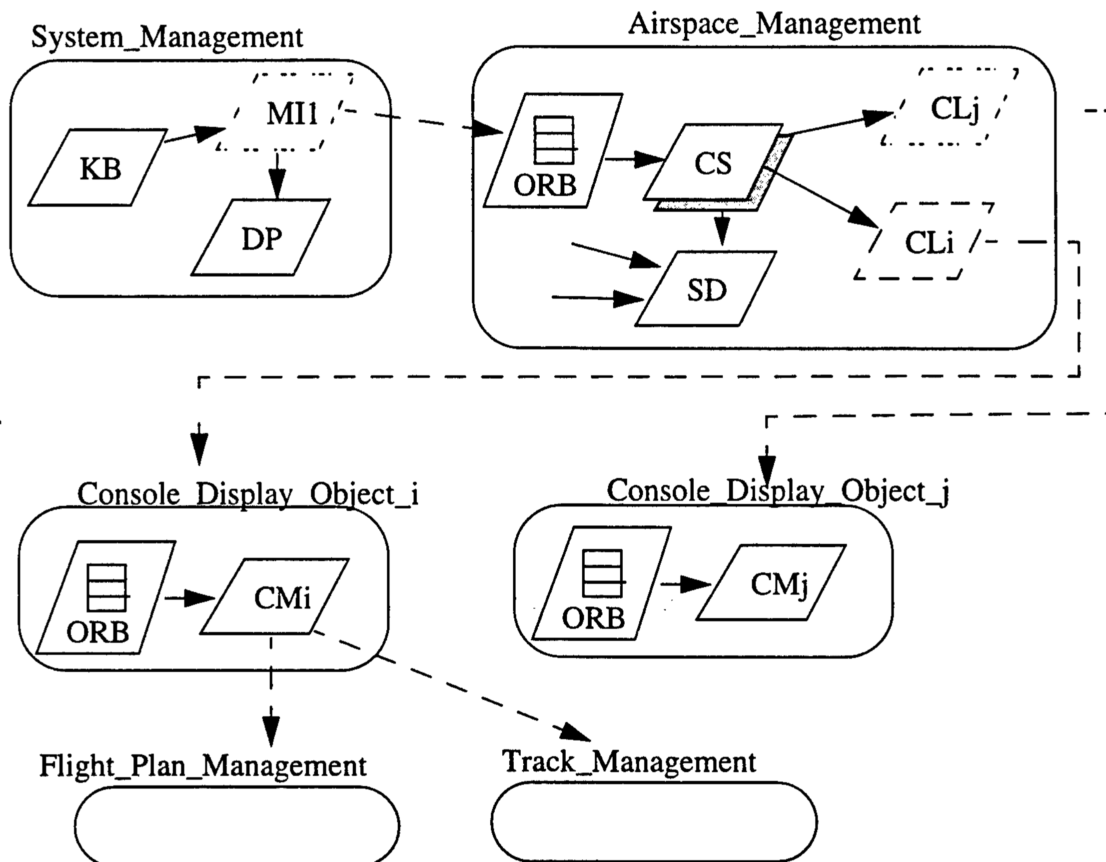


Figure 74: Summary of Resectorization "String" of Computation

Table 9 summarizes the schedulability expressions for the constituents of the resectorization string of computation. For each object, the schedulability expressions are labelled as preemption, execution, or blocking.

Note that the above expression for *System_Management* is slightly different than the one shown at the bottom of Table 6. Here, we assume that the method invocation does not time out, and hence we do not show a time-out term in the expression. Second, the expression in Table 9 represents *System_Management*'s contribution to end-to-end latency, whereas the expression in Table 6 modeled KB's latency. The time-out term also does not appear in the expression for *Airspace_Management.*

Subsystem		Schedulability Expression
System_Management	Preempt	$n_{inv} [C_{MI\text{-}end} + C_{DP}]$
	Exec	$C_{KB} + C_{MI\text{-}start} + C_{MI\text{-}end} + C_{DP}$
Airspace_Management	Exec	$C_{CS} + C_{SD\text{-}read} + n_{combine} [C_{CI\text{-}start} + C_{CI\text{-}end}]$
	Block	$N_{Flights}B_{ORB} + B_{SD\text{-}write}$
Console_Display_Object_i	Exec	$C_{CMi} + C_{TM} + C_{FP\,M} + C_{LD}$
	Block	$(N_{Flights}/n_{consoles})B_{ORB} + B_{TM} + B_{FPM} + B_{LD}$

Table 9: Schedulability Model for Resectorization

The end-to-end latency for resectorization is computed by adding together the latencies due to *System_Management*, *Airspace_Management*, and *Console_Display_Object_i*. In our case, this means simply adding together the expressions in Table 9. However, in most cases, accounting for preemption is much more involved.

6.3.3.2.5 Performance Observations for CORBA-Based Design

There are several observations that we can make simply by examining the expressions in Table 9, without having to plug numbers into the expressions:

1. Blocking could be a major contributor to resectorization latency. In particular, the potential blocking due to the assumed FIFO queue in the ORB could be significant.
2. There is a significant amount of data marshalling and unmarshalling in the resectorization string of computation. If the computational cost for marshalling and unmarshalling is high, it could contribute to end-to-end latency. For homogeneous systems, this cost would be unnecessary overhead. For heterogeneous systems, the data transformations are necessary with the cost being incurred either by CORBA or the application.
3. The synchronous nature of method invocations combined with the lack of "timed method invocations" resulted in the need for dynamic thread creation and termination in several places. The cost of this service needs to be examined to ensure that it is not too high.
4. Priority assignment for threads controlling shared resources is very important. This point is subtle, since we assigned priorities that achieve a minimal amount of blocking. However, when assigning SD in *Airspace_Management*, for example, a lower priority could result in a significant level of blocking.

6.3.4 Performance Using POSIX

In this section, we look at a realization of the scenario shown in Section 6.3.2 using a POSIX-based architecture.

6.3.4.1 Scenario Realization

A high-level realization of the resectorization scenario using POSIX is described below:

1. The system management operator initiates a request to combine two (or more) sectors, which results in a message being sent from *System_Management* to *Airspace_Management*.
2. *Airspace_Management* needs to validate the request. (Determine if the sector number is valid, sector state is valid, etc.)
3. *Airspace_Management* sends messages to each of the *Console_Display_Objects* whose geometry has changed.
4. The active sector (sector into which other sectors are being joined) gets list data either from the consoles on which the inactive sectors reside, or by reconstructing list data from locally stored track and flight data.
5. *Airspace_Management* adjusts its view of active sectors.

This high-level realization is discussed in more detail below. It is subdivided into three main parts:

- System management operator initiates a resectorization request
- *Combine_Sector* in airspace management is invoked
- Airspace management sends message to console displays

The discussion that follows is less detailed than the CORBA realizations. Since the realizations are not substantially different than those for CORBA, we focus on the differences between CORBA and POSIX.

6.3.4.1.1 System Management Operator Initiates a Resectorization Request

The POSIX process architecture is slightly different than CORBA's, the difference motivated by the desire to exploit the asynchronous send facility of POSIX. (Recall that CORBA method invocations are synchronous.)

As for CORBA, KB is responsible for parsing commands. The command is sent (via a local synchronous interthread communication facility) to the sending (Snd) thread, which in turn asynchronously sends a message to a receiving endpoint of *Airspace_Management*. The mes-

sage contains all of the information needed to direct *Airspace_Management* to perform a resectorization. Embedded in the message is a message label, which will be used by *Airspace_Management* when it ultimately replies that resectorization is complete or has failed. This process is illustrated in Figure 75.

Snd consists of a tight loop, basically waiting for KB to send it a command. It then sends a message asynchronously to another subsystem (in this case *Airspace_Management*), starts a timer, and returns to the top of the loop. The receiving thread, Rcv, waits for labelled messages or time-out signals. Rcv has registered for a set of specified message labels at system initialization. For each message sent by Snd, Rcv will receive either a labelled message in response or a time-out (associated with the same label). Rcv responds by passing this information to DP, which displays an appropriate message on the system console.

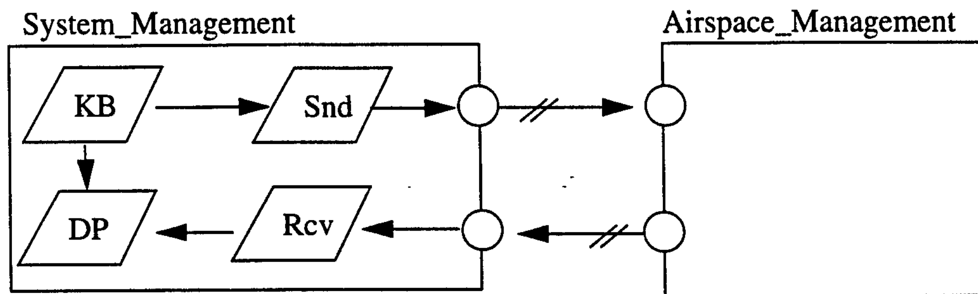


Figure 75: System_Management Threads When Using POSIX

6.3.4.1.1.1 Scenario and Requirement Refinements

Note that no new scenario/requirement refinements were discovered relative to the CORBA equivalent of *System_Management*.

6.3.4.1.1.2 Design Constraints and Assumptions

The following design constraints and assumptions became evident as a consequence of realizing this scenario using POSIX:

- Assume the ability to send messages asynchronously.
- Assume the ability to associate a time-out with a message.

6.3.4.1.1.3 Design Considerations

The following design considerations have an impact on the subsequent performance analysis:

- Since KB and DP will initiate and display information involving all of the subsystems and thus communicate with all of the subsystems, we chose to separate sending and receiving services into their own threads, respectively.

- Using asynchronous communications with other subsystems will allow KB to have a quick response time.

6.3.4.1.2 Combine_Sector in Airspace Management Is Invoked

Threads are shown in Figure 76 for *Airspace_Management* for a POSIX implementation: CS, SD, and OP. The thread responsible for combining sectors (CS) blocks on labelled messages (relevant to resectorization) sent to the receiving endpoint of *Airspace_Management*. CS uses the same protocol discussed in the CORBA implementation to effect a resectorization between two or more consoles. Sector data are still managed by *Airspace_Management* and can be shared by multiple threads; hence, mutual exclusive access is enforced by SD. There are also likely to be other threads for handling other processing (OP) for which *Airspace_Management* is responsible. We call this out explicitly to emphasize that endpoints will be shared with other threads, and all threads will wait for the appropriate labelled messages using POSIX facilities.

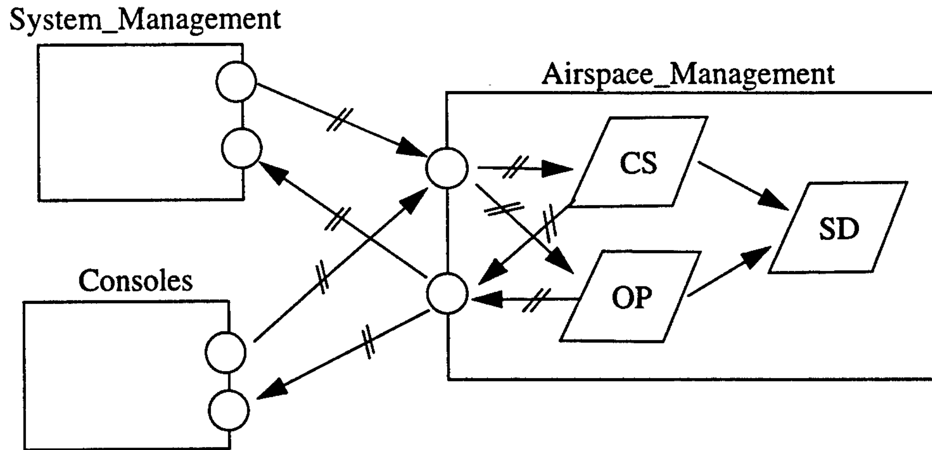


Figure 76: Airspace_Management Threads When Using POSIX

6.3.4.1.2.1 Scenario and Requirement Refinements

Note that no new scenario/requirement refinements were discovered relative to the CORBA equivalent of *System_Management*.

6.3.4.1.2.2 Design Constraints and Assumptions

The following design constraint and assumption became evident as a consequence of realizing this scenario using POSIX:

- Assume minimal overhead in handling a priority queue of prioritized messages.

6.3.4.1.2.3 Design Considerations

The following design consideration has an impact on the subsequent performance analysis:

- Assume that the priority of OP is less than the priority of CS.

6.3.4.1.3 Airspace Management Sends Message to Console Displays

The main difference between the POSIX and CORBA implementations is that in the POSIX implementation, the consoles have a local copy of all flight-plan and track data. This allows list data to be reconstructed using local data. OP represents another thread which is responsible for receiving and updating the local data stores. Note that OP possibly fields interrupts due to the multicasting of track data. The relevant threads for this case are shown in Figure 77.

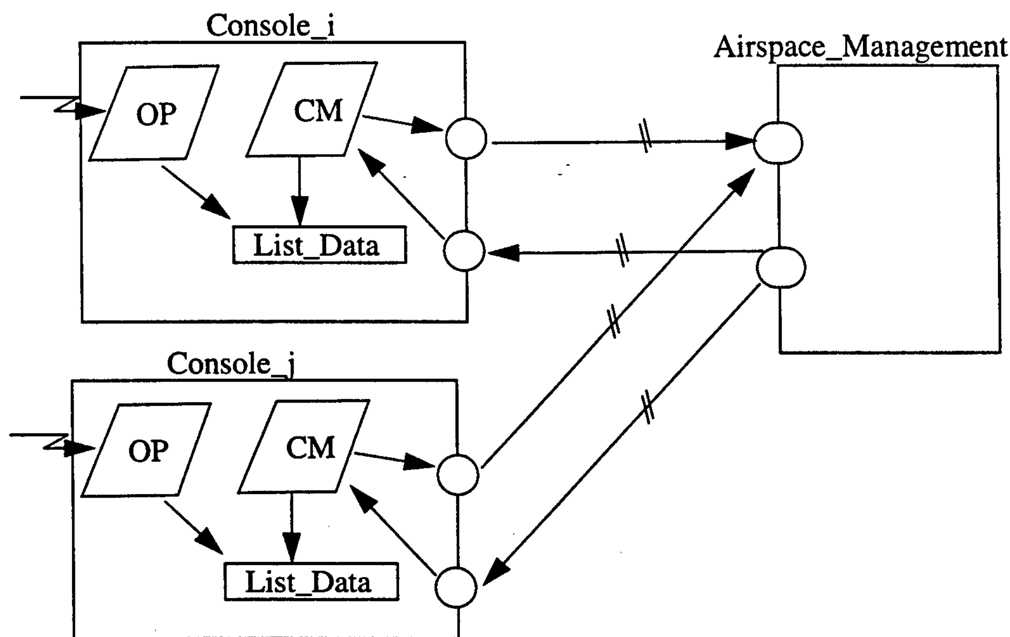


Figure 77: Console Threads When Using POSIX

6.3.4.2 Performance Modeling

We develop performance models for the POSIX realizations in this section. Continue to be aware that the performance models are representative of what can be done, but are by no means comprehensive.

Again, we consider the scenario in three stages:

- System management operator initiates a resectorization request
- *Combine_Sector* in airspace management is invoked
- Airspace management sends message to console displays

After the three stages are discussed, end-to-end latency for the scenario is discussed.

6.3.4.2.1 System Management Operator Initiates a Resectorization Request

The schedulability model for *System_Management* using POSIX is very similar to the model when CORBA is used. This should not be surprising since the CORBA concurrency architecture for *System_Management* using CORBA was constructed to emulate asynchronous communication.

<table>
<tr><th>Process / Thread</th><th>Priority</th><th>Description of Computation</th><th>Schedulability Expression</th></tr>
<tr><td>KB</td><td>10</td><td>KB computation
IPC to Snd</td><td rowspan="2">$C_{KB} + C_{Snd}$</td></tr>
<tr><td>Snd</td><td>20</td><td>Async send</td></tr>
<tr><td>Rcv</td><td>20</td><td>Receive message and process
or
Receive time-out signal and process
IPC to DP</td><td rowspan="2">$\max(C_{TO}, C_{Rcv}) + C_{DP}$</td></tr>
<tr><td>DP</td><td>15</td><td>Display status</td></tr>
<tr><td colspan="4">"UI latency" = $C_{KB} + C_{Snd} + n_{inv}[\max(C_{TO}, C_{Rcv}) + C_{DP}]$</td></tr>
</table>

Table 10: Schedulability Model for Keyboard "String" Using POSIX

While the schedulability expressions for CORBA and POSIX appear to be very similar, some possible differences are highlighted by the difference in the description of the computations shown in Table 6 and Table 10, respectively. In particular, the CORBA implementation must incur the overhead of thread creation and deletion to achieve the effect of asynchronous communication. Moreover, the overhead due to the marshalling and unmarshalling of data in CORBA should be checked to see if it is significant.

6.3.4.2.2 Combine_Sector in Airspace Management Is Invoked

The main difference between the schedulability expressions for the CORBA and POSIX implementations of *Airspace_Management* are due to the effect of track processing on each. In the CORBA implementation, all of the track data were routed through *Airspace_Management*, whereas in the POSIX implementation the data were multicasted to all of the consoles directly from *Track_Management*. As shown in Table 11, the blocking term that was present for CORBA is gone for POSIX. Moreover, through priority queues associated with endpoints and assigning the appropriate priority to threads responsible for other processing (OP), the preemptive and blocking effects due to other processing are easily controlled.

Process / Thread	Priority	Description of Computation	Schedulability Expression
CS	20	Start combination Possibly block on SD	$C_{CS} + B_{SD\text{-}write} + C_{SD\text{-}read}$
SD	35	Read or write sector data	
CS-Snd	20	Start timer Async send	$C_{CS\text{-}send} + \max(C_{TO}, C_{CS\text{-}rvc})$
CS-Rcv	20	Receive message or time-out	
OP	10	Other lower priority processing	
n/a	n/a	Latency for console operations	$\max(L_{CLi}, L_{CLj})$

"Combine Sectors Latency" =

$$C_{CS}+B_{SD\text{-}write} + C_{SD\text{-}read} + n_{combine} [C_{CS\text{-}send} + \max(C_{TO}, C_{CS\text{-}rcv})] + \max(L_{CLi}, L_{CLj})$$

Table 11: Schedulability Model for Combine_Sectors "String"

6.3.4.2.3 Airspace Management Sends Message to Console Displays

The main difference between CORBA and POSIX implementations of the console is due to the use of multicast for the POSIX implementation. This difference manifests itself primarily in two places.

Consoles will be keeping a local copy of track data and hence will be receiving all track data, not just the track data they display. Depending on the hardware configuration, the arrival of

each track could cause a processor interrupt and thus add a preemption term to the schedulability expression. The computing associated with inserting the data into the track database can be relegated to a lower priority and hence will not affect resectorization. (These computations are shown in Table 12.)

Consoles can reconstitute list data locally or acquire the data from the subordinate console; in either case, the blocking due to the FIFO queue in the ORB is not experienced by the POSIX implementation.

Process / Thread	**Priority**	**Description of Computation**	**Schedulability Expression**
Interrupt	50	Interrupt due to multicast of track data	$N_{Flights}C_{Interrupt}$
OP	10	Local processing of track data	$N_{Flights}C_{OP}$
CM	20	Resectorization processing	C_{CMi}
LD	n/a	Blocking due to LD Reconstruct list data	$C_{LD} + B_{LD}$
"Assign Sector Latency" = $N_{Flights}C_{Interrupt} + C_{CMi} + C_{LD} + B_{LD}$			

Table 12: Schedulability Model for Assign_Sector in Console Object

6.3.4.2.4 End-to-End Latency for Resectorization

Figure 78 summarizes the resectorization string of computation for POSIX.

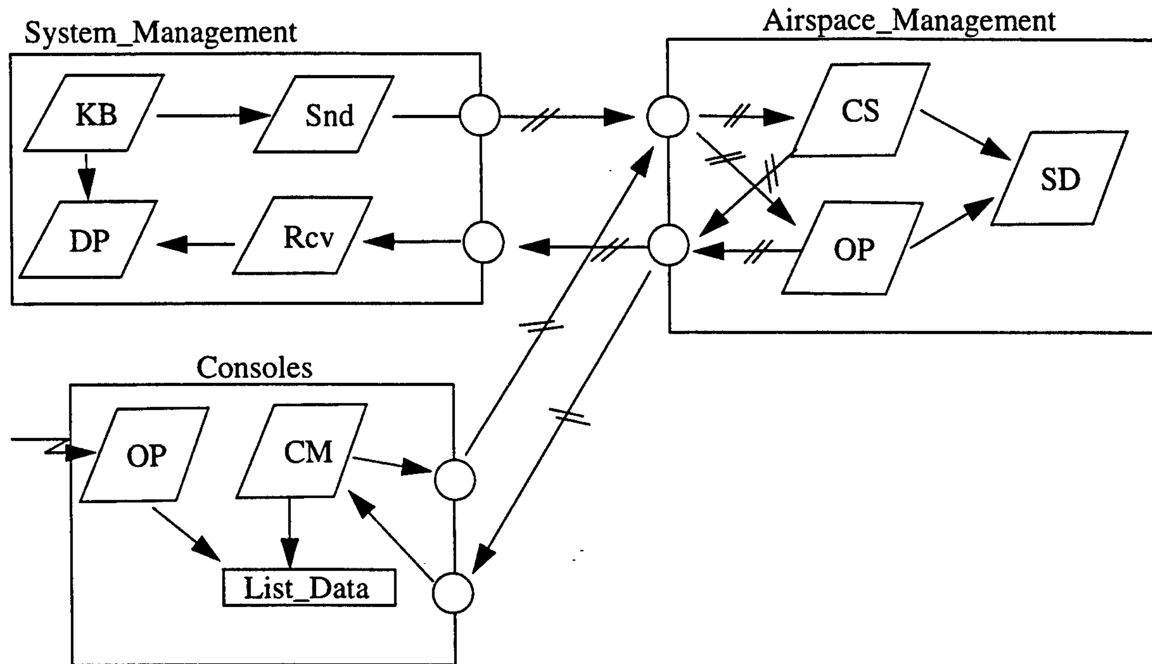


Figure 78: Summary of Resectorization "String" of Computation Using POSIX

Table 13 summarizes the schedulability expressions for the constituents of the resectorization string of computation. For each subsystem, the schedulability expressions are labelled as preemption, execution, or blocking.

Subsystem		Schedulability Expression
System_Management	Preempt	$n_{inv} [C_{Rcv} + C_{DP}]$
	Exec	$C_{KB} + C_{Snd} + C_{Rcv} + C_{DP}$
Airspace_Management	Exec	$C_{CS} + C_{SD\text{-}read} + n_{combine} [C_{CS\text{-}send} + C_{CS\text{-}rcv}]$
	Block	$B_{SD\text{-}write}$
Console_i	Preempt	$N_{Flights} C_{Interrupt}$
	Exec	$C_{CMi} + C_{LD}$
	Block	B_{LD}

Table 13: Schedulability Model for Resectorization Using POSIX

As for the CORBA implementation, note that the above expression for *System_Management* is slightly different than the one shown at the bottom of Table 11. Again, we assume that no time-out occurs, and hence, we do not show a time-out term in the expressions. Secondly, the expression in Table 13 represents *System_Management*'s contribution to end-to-end latency, whereas the expression in Table 11 modeled KB's latency.

The end-to-end latency for resectorization is computed by adding together the latencies due to *System_Management*, *Airspace_Management*, and *Console_i*. In our case, this means simply adding together the expressions in Table 13. However in most cases, accounting for preemption is much more involved.

6.3.4.2.5 Performance Observations: CORBA Vs. POSIX Designs

Table 14 shows the schedulability expressions for the CORBA and POSIX implementations of *System_Management (SM)*, *Airspace_Management (AM)*, *Console_i (CDi)* and *Console_j (CDj)*.

		Schedulability Expression	
Subsystem		**CORBA**	**POSIX**
SM	Preempt	n_{inv} [$C_{MI\text{-}end} + C_{DP}$]	n_{inv} [$C_{Rcv} + C_{DP}$]
	Exec	$C_{KB} + C_{MI\text{-}start} + C_{MI\text{-}end} + C_{DP}$	$C_{KB} + C_{Snd} + C_{Rcv} + C_{DP}$
AM	Exec	$C_{CS} + C_{SD\text{-}read} +$ $n_{combine}$ [$C_{CI\text{-}start} + C_{CI\text{-}end}$]	$C_{CS} + C_{SD\text{-}read} +$ $n_{combine}$ [$C_{CS\text{-}send} + C_{CS\text{-}rcv}$]
	Block	$N_{Flights}B_{ORB} + B_{SD\text{-}write}$	$B_{SD\text{-}write}$
CDi	Preempt		$N_{Flights}C_{Interrupt}$
	Exec	$C_{CMi} + C_{TM} + C_{FP\,M} + C_{LD}$	$C_{CMi} + C_{LD}$
	Block	$(N_{Flights}/n_{consoles})B_{ORB} + B_{TM} + B_{FPM} +$ B_{LD}	B_{LD}

Table 14: Schedulability Comparison for Resectorization

There are several observations that we can make simply be examining the expressions in Tables 13 and 14:

- The ability to use asynchronous communications seems to have simplified the design.
- The elimination of FIFO queues and the use of priority queues lessens blocking. One needs to check the efficiency of priority queues and the communication facilities in the POSIX implementation to ensure that the apparent benefits are being achieved.

- Multicasting and storing track data locally at the consoles introduced some overhead due to the processing of track-data interrupts. The benefit gained is not having to acquire the data remotely, thus allowing for quicker console restarts.

Many of the assumptions that we made concerning CORBA can be traced to the current specification for CORBA. For example, the basic interaction in CORBA is defined as a unicast, synchronous remote procedure call. In addition, the lack of priorities in CORBA has a consequence of FIFO queuing method invocations.

6.4 Modifiability Assessment

The modifiability assessment of the two candidate designs—CORBA and POSIX—will be done using the SAAM, which feeds into the overall ATAM, as described in Section 6.1.1.

6.4.1 Brief Description of SAAM

The SAAM method consists of the following steps (a subset of the ATAM's steps):

1. *scenario elicitation*: Anticipated uses of, and changes to, the system are described. There is an important distinction between scenario types that we introduce at this point. Recall that a scenario is a brief description of some anticipated or desired use of a system. The system may directly support that scenario, meaning that anticipated use requires no modification to the system for the scenario to be performed. This would usually be determined by demonstrating how the existing architecture would behave in performing the scenario (rather like a simulation of the system at the architectural level). If a scenario is not directly supported, there must be some change to the system that we could represent architecturally. This change could be a change to how one or more components perform an assigned activity, the addition of a component to perform some activity, the addition of a connection between existing components, or a combination of these.
2. *architectural view description*: In the case of SAAM, the view of interest is a developer's view, that emphasizes the code units to be modified.
3. *scenario realization*: Scenarios are mapped onto the architectural description. For each indirect scenario, the changes to the architecture that are necessary for it to support the scenario must be listed, and the cost of performing the change must be estimated. A modification to the architecture means that either a new component or connection is introduced, or an existing component or connection requires a change in its specification.
4. *assessment of the design*: For each indirect scenario, the effect or set of changes that the scenario has on the architecture is described. A weighting of the change's difficulty also accompanies this stage. Usually, this weighting is coarse grained, based on the understanding of the architecture—it may be nothing more than an order of magnitude estima-

tion (1 day, 10 days, 100 days, etc.), or it may be more precise if sufficiently detailed design information is available to make the assessment of difficulty confidently. A tabular summary is especially useful when comparing alternative architectural candidates because it provides an easy way to determine which architecture better supports a collection of scenarios.

When two or more indirect task scenarios require changes to a single component of a system, they are said to *interact* in that component. Scenario interaction is important to highlight because it exposes the allocation of functionality to the product's design. The interaction of semantically unrelated scenarios explicitly shows which system modules are computing semantically unrelated functions. Areas of high scenario interaction reveal potentially poor separation of concerns in a system component. Thus, areas of scenario interaction indicate where the designer should focus subsequent attention. The amount of scenario interaction is related to metrics (such as structural complexity, coupling, and cohesion). Therefore, it is likely to be strongly correlated with the number of defects in the final product.

Finally, a weight is assigned to each scenario and the scenario interactions in terms of their relative importance. The weighting should be used to determine an overall ranking. The purpose of assigning weights is to resolve the situation in which the first candidate architecture scores well on half the scenarios, and the second candidate architecture scores better on the other half. Assigning weights is a subjective process involving all of the stakeholders in the system. Rather than offering a single architectural metric, SAAM produces a collection of small metrics, a set of per-scenario analyses. Given this set of small metrics, SAAM can be used (and, in fact, was developed with the intent) to compare competing architectures on a per-scenario basis. It is left to the users of SAAM to determine which scenarios are most important to them so that they can resolve cases in which the candidates out-score one another on different scenarios. The process of performing a SAAM analysis has also been used to gain a high-level understanding of the competing architectures; this high-level understanding, rather than just a scenario-based table, is useful for performing comparative analysis.

6.4.2 Modifiability Using CORBA

We consider three modifiability scenarios, a subset of the scenarios presented in Section 6.2, in this analysis:

1. dynamic sector boundaries
2. live insertion of consoles
3. have all consoles aware of all data

Every architecture partitions possible changes into three categories: *local, non-local,* and *architectural.* A local change can be accomplished by modifying a single component. A non-local change requires multiple component modifications, but leaves the underlying architecture intact. An architectural change affects the fundamental ways in which the components interact with each other—the style of the architecture—and frequently requires changes all over the system. Obviously, local changes are the most desirable; therefore, an effective architecture is one in which the most likely changes are also the easiest to make.

6.4.2.1 Scenario Realization and Refinement

When considering the effects of the changes implied by scenarios 1 and 2, we will see that those scenarios both have non-local effects for the CORBA design. But scenario 3 for the CORBA case has architectural implications; it changes the topology, data, and control flow of the entire system. Because of this, we will consider that change separately from the others, which leave the underlying architecture intact.

6.4.2.2 Scenario 1: Dynamic Sector Boundaries in CORBA

Supporting the ability to have a controller dynamically (through direct manipulation of the user interface) redefine sector boundaries involves the changes described in Table 15.

Description of Modification	Estimate of Effort (person weeks)
System Management GUI modifications (to support user interaction)	2.0
System Management IDL modifications (to support new messages to System Management Console)	0.5
Additional System Management logic changes (to support new user interaction/interfaces)	5.0
Additional Airspace Management logic changes (to support new data filtering/forwarding processing)	6.0
Airspace Management Object IDL modifications (to support new messages to System Management Console)	0.5
Total	14.0

Table 15: Modifications Required to Satisfy Dynamic Sector Boundaries in CORBA

In Figure 79, these changes are mapped onto the system's architectural representation as indicated by the "1" annotating objects and their interfaces.

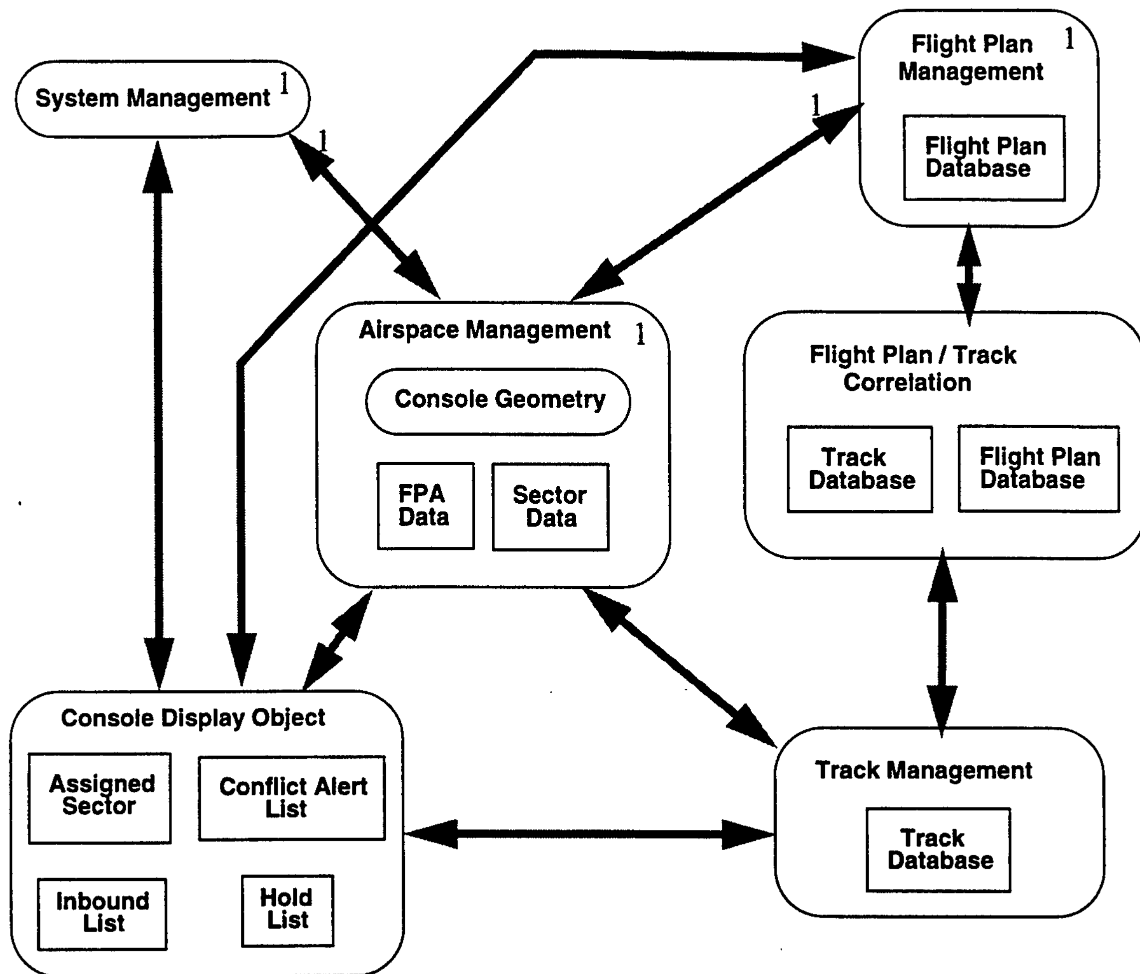


Figure 79: CORBA Design with Realization of Scenario 1

6.4.2.3 Scenario 2: Live Insertion in CORBA

This scenario describes the ability of the system to contain a number of consoles to serve as a backup for radar display consoles, where the system has the ability to insert a console into the network during operational conditions ("live insertion").

The changes required to satisfy this scenario are described in Table 16.

Description of Modification	Estimate of Effort (person weeks)
GUI modifications for Console Display Object (for when it first becomes active, to support configuration)	1.0
Modifications to the server object that will provide operational software	3.0
Modifications to Flight Plan Management and Track Management, to send data to new consoles	6.0
Total	10.0

Table 16: Summary of Changes to Support Live Insertion in CORBA

In Figure 80, these changes are mapped onto the system's architectural representation from Figure 81 as indicated by the "2" annotating objects and their interfaces.

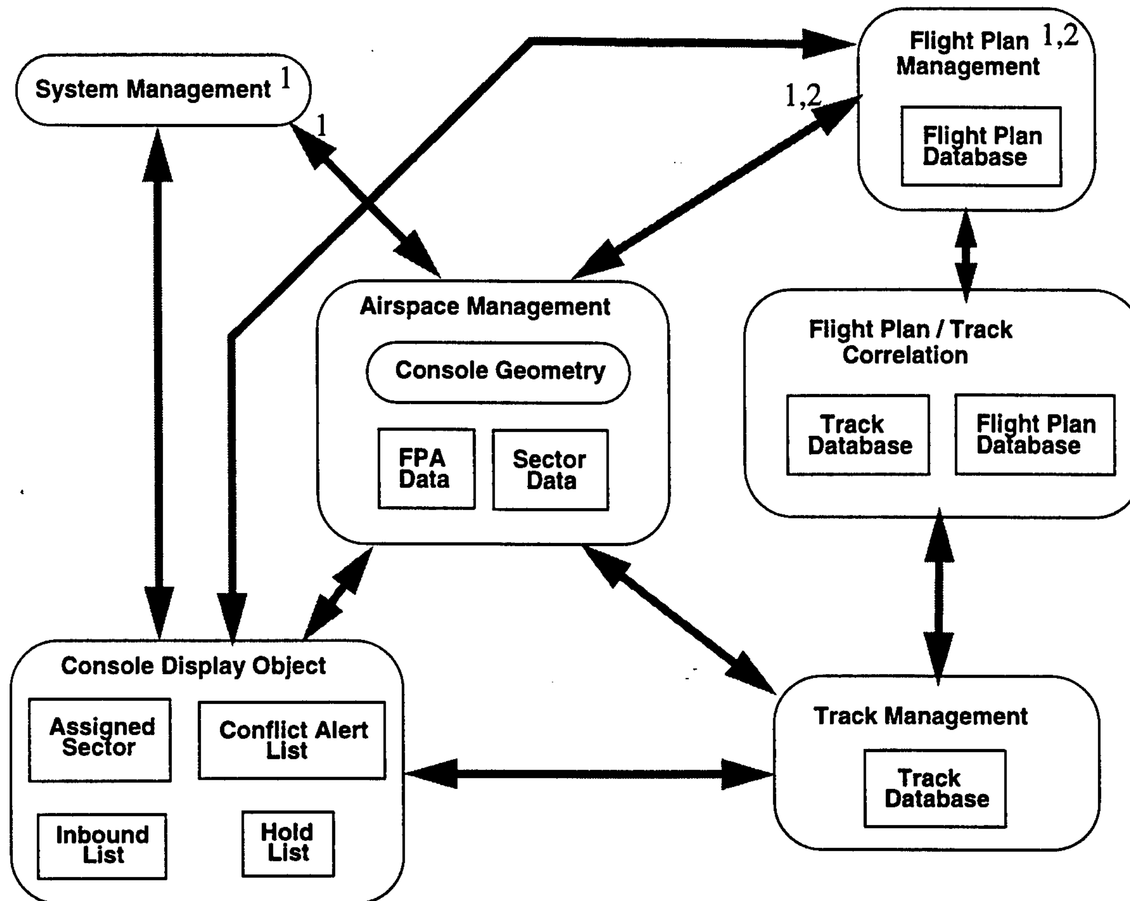


Figure 80: CORBA Design with Realization of Scenarios 1 and 2

6.4.2.4 Scenario 3: Have All Consoles Aware of All Data in CORBA

Supporting the ability to have all consoles aware of all data means that any console could act as a backup for any other, or could include any sectors with only minimal changes (in terms of the data that it needs to be aware of). This modification involves the changes described in Table 17.

Description of Modification	Estimate of Effort (person weeks)
Remove airspace manager	1.0
Change track manager to transmit data to all consoles	1.0
Change flight plan manager to transmit data to all consoles	1.0
Move console geometry logic into console display objects	3.0
Move controlling the sector and FPA assignments to system management	1.0
Total	7.0

Table 17: Modifications Required to Satisfy "Aware Consoles" in CORBA

These changes are architectural. That is, they dramatically alter the system's communication topology, its allocation to hardware, the flow of data control and control in the system, and the allocation of functionality to structure. The new architecture resulting from the application of scenario 3 would appear as shown Figure 81.

In addition to these structural changes, communication will be radically altered (in that many more messages will be flowing around the system, since now every console will be receiving each message). If there are n consoles and m track or flight-plan messages per second, communication goes from approximately $2m$ to nm cost, assuming that this is done naively (by simply sending the same message n times, one to each console).

Regardless of how this is achieved, it will have enormous implications for the system's performance; therefore the performance models built in Section 6.3 will have to be revisited. This is an example of how attributes interact and how this interaction is modeled in an ATAM analysis.

6.4.3 Modifiability Using POSIX

For the consideration of the POSIX design, we once again consider three modifiability scenarios, a subset of the scenarios presented in Section 6.2:

1. dynamic sector boundaries
2. live insertion of consoles
3. have all consoles aware of all data

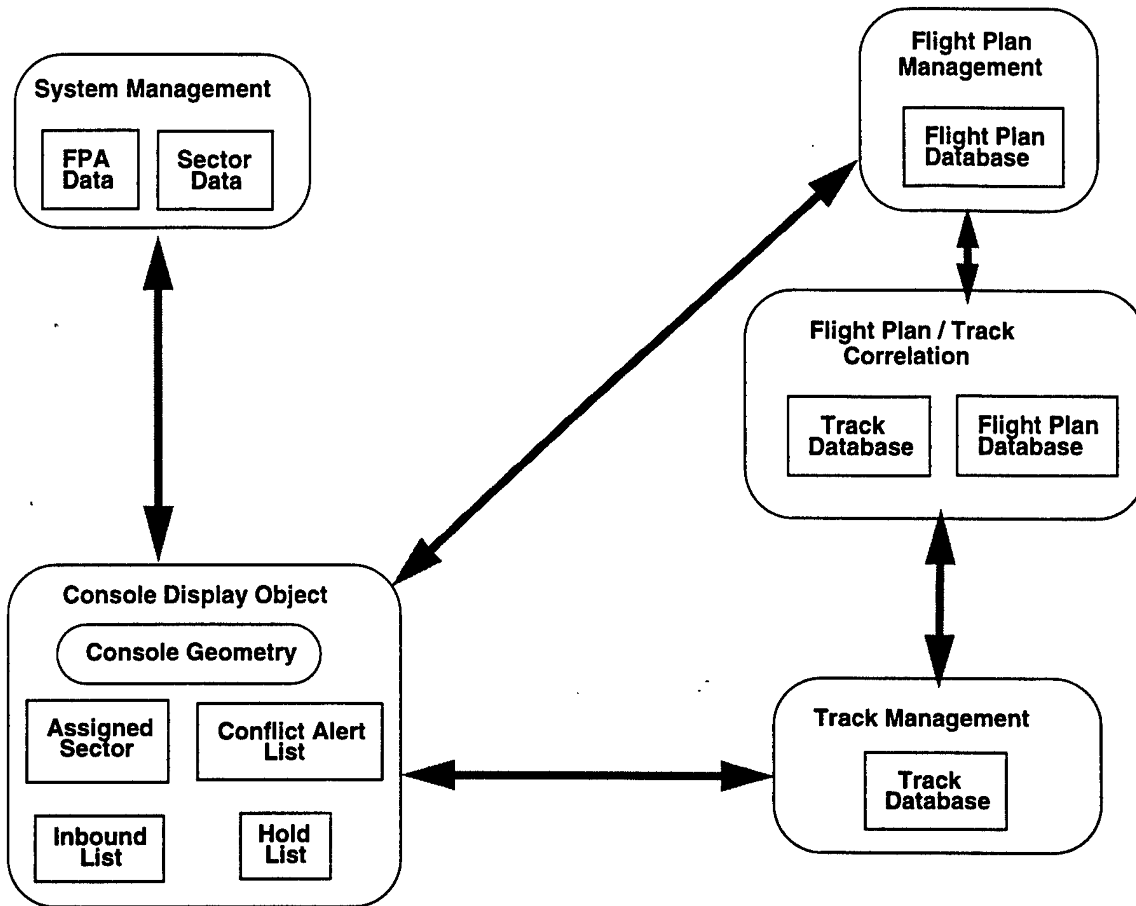


Figure 81: CORBA Design with Realization of Scenarios 1, 2 and 3

6.4.3.1 Scenario Realization and Refinement

When considering the effects of the changes implied by scenarios 1 and 2, we will see that those scenarios both have non-local effects for the POSIX design. But scenario 3 for the POSIX case has no implications, because this is the way that the system already operates. In SAAM terms, it is a *direct* scenario.

6.4.3.2 Scenario 1: Dynamic Sector Boundaries in POSIX

Supporting the ability to have a controller dynamically (through direct manipulation of the user interface) redefine sector boundaries involves the changes shown in Table 18.

Description of Modification	Estimate of Effort (person weeks)
System Management GUI modification (to support user interaction)	2.0
Modify message formats for System Management including marshalling	2.0
Additional System Management logic changes (to support new user interaction/interfaces)	5.0
Modify message formats for Airspace Management (including marshalling)	2.0
Additional Airspace Management logic plus modifications	4.0
Total	15.0

Table 18: Modifications Required to Satisfy Dynamic Sector Boundaries in POSIX

In Figure 82, these changes are mapped onto the system's architectural representation as indicated by the "1" annotating objects and their interfaces.

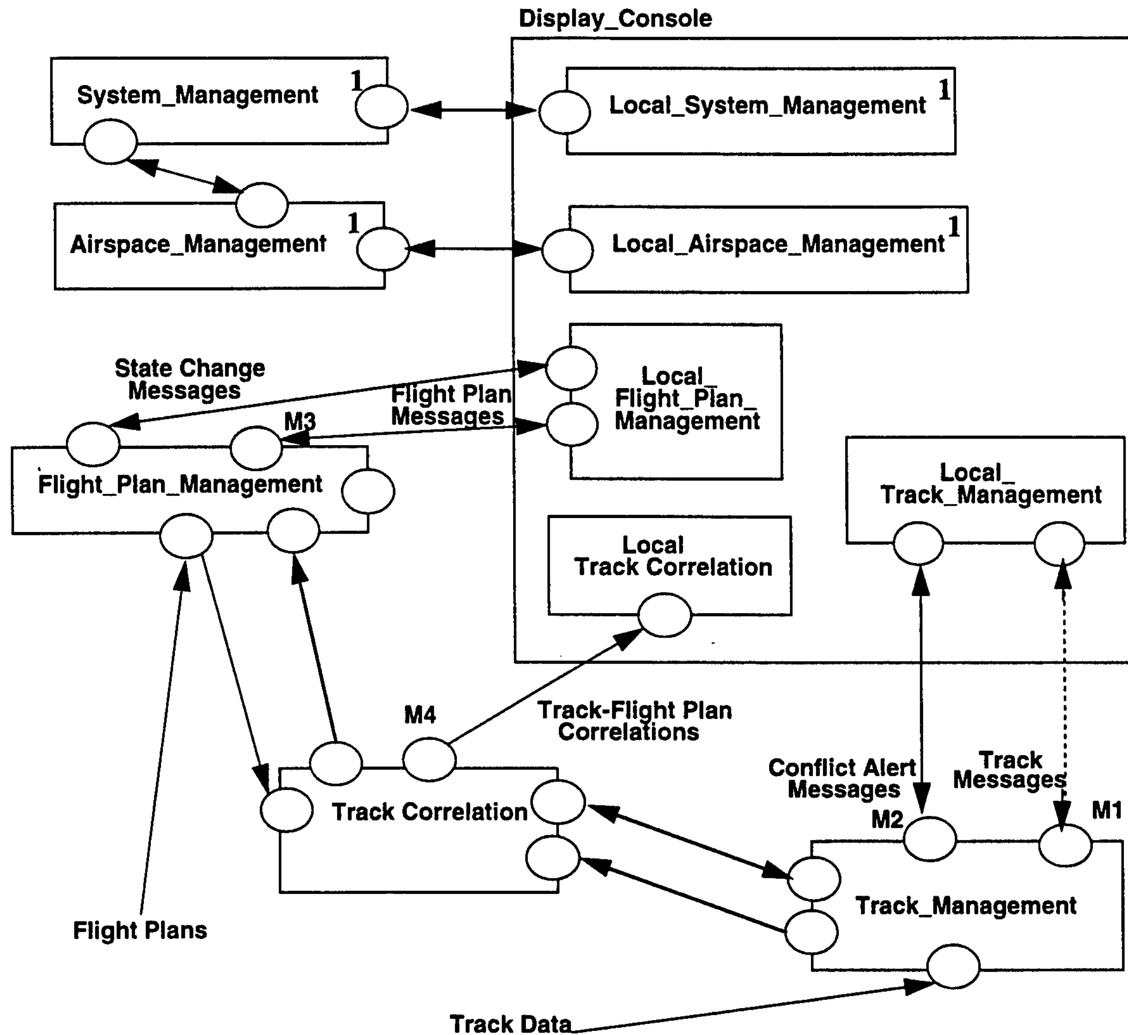


Figure 82: POSIX Design with Realization of Scenario 1

6.4.3.3 Scenario 2: Live Insertion in POSIX

This scenario describes the ability for the system to contain a number of consoles to serve as a backup for radar display consoles, where the system has the ability to insert a console into the network during operational conditions ("live insertion"). The changes required to satisfy this scenario are described in Table 19.

Description of Modification	Estimate of Effort (person weeks)
GUI modifications for Console Display Object (for when it first becomes active, to support configuration)	1.0
Modifications to the server object that will provide operational software	3.0
Modifications to Flight Plan Management and Track Management, to send data to new console	6.0
Total	10.0

Table 19: Summary of Changes to Support Live Insertion in POSIX

In Figure 83, these changes are mapped onto the system's architectural representation from-Figure 82 as indicated by the "2" annotating objects and their interfaces.

6.4.3.4 Scenario 3: Have All Consoles Aware of All Data in POSIX

This scenario is direct. This is how the system operates now, so no changes are needed.

6.4.4 Comparing CORBA to POSIX

The purpose of this example assessment was not to rank the CORBA or POSIX designs for their quality, but rather to provide an example of how the assessment proceeds, what the implications of an assessment are, and how assessment results from the consideration of one quality feed into another quality.

We have not considered enough scenarios here to provide a comprehensive analysis; even if we had considered enough scenarios, these scenarios have not been elicited from the true stakeholders of the system, which they must be for the analysis to be valid.

The purpose of this exercise, then, is to show how scenarios are realized on a system's architecture to shed light on the changes necessary to meet an anticipated change to the system (an indirect scenario), or if a scenario can be met by the system as is (a direct scenario). Obviously, all other things being equal, one prefers direct scenarios to indirect ones with respect to any design. Among indirect scenarios, we look for places where many different scenarios coalesce in a single structure. This indicates a potential area of high complexity within the architecture, because this structure is the site of many unrelated changes.

For example, flight-plan management, in Figure 80, is the site of changes relating to both scenario 1 and 2 for the CORBA design. This might be an early indication of a problem area. But only a larger number of scenarios, elicited from a population of the system's stakeholders, will meaningfully uncover such areas.

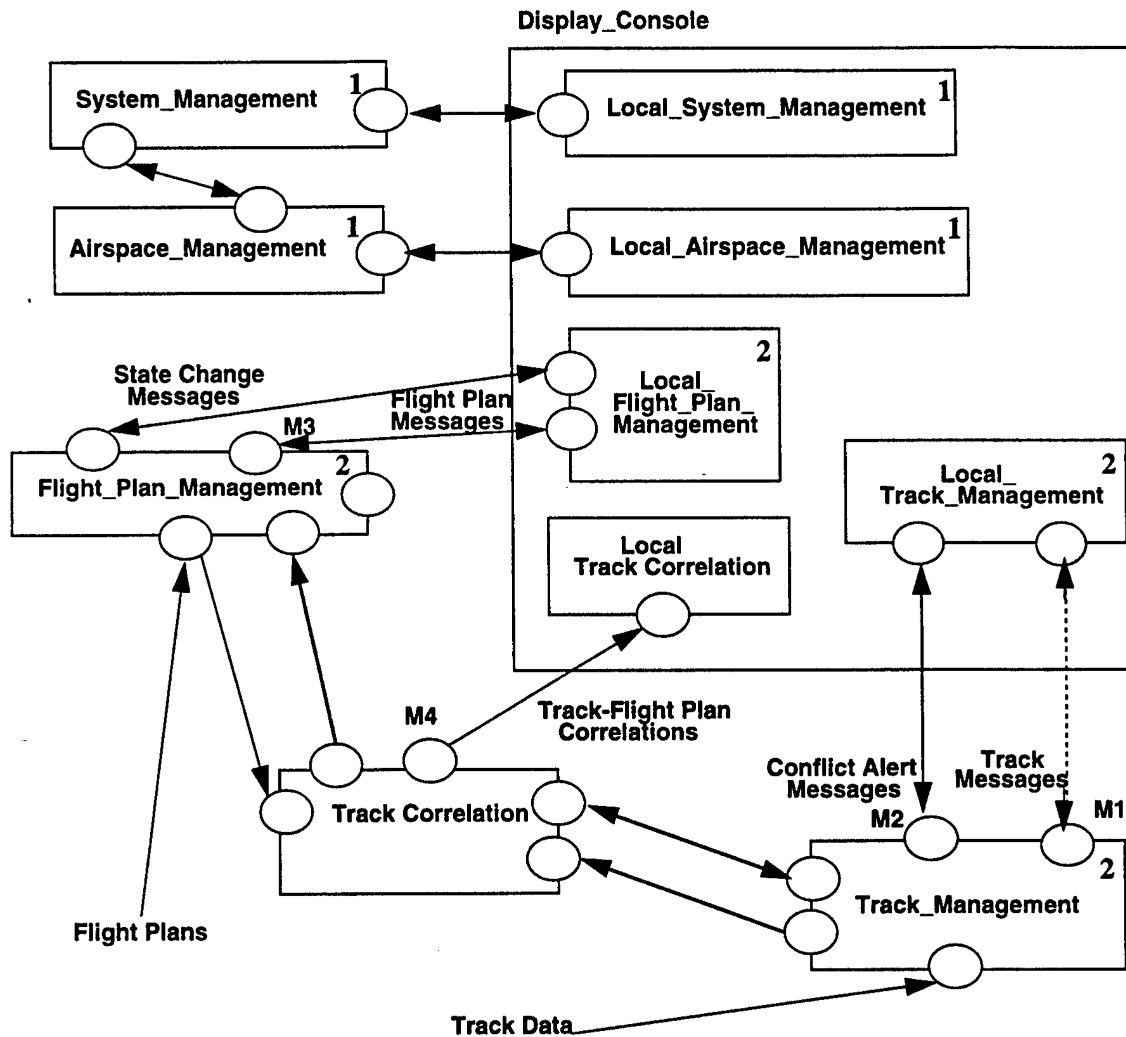


Figure 83: POSIX Design with Realization of Scenario 2

7 Summary

The purpose of this report was to identify issues regarding the use of CORBA and POSIX.21 in a large distributed system. In particular, the function of sector combination in the context of an FAA En Route architecture was addressed. The ability to reconfigure a sector geometry (through sector combination, for example) is a current En Route capability, and extensions of this capability are believed to apply to free flight.

The approach of this report was to

- develop designs for resectorization using CORBA and POSIX.21
- assess the designs using an architectural trade-off analysis (ATA) approach, focusing on modifiability and performance

The following points are relevant to the summary of the designs:

- The CORBA design was initially based on a maximal object principle and then iteratively refined. The refinement was motivated by performance and fault-tolerance considerations. The final design, for the resectorization problem, had a small number of large-grained objects.
- The POSIX.21 design focused on communication needs for resectorization. This led to a functional partitioning, and focused on data considerations and mechanisms to support data transfer.

The major difference between the CORBA and POSIX.21 designs was due to the use of different communication models. Some of the consequences of this appear in central versus local track management, fault tolerance, and the ability to localize changes to consoles, not other functions.

The following points are relevant to the assessment of the designs:

- modifiability: Both designs were believed to be equally modifiable with respect to the chosen scenarios.
- performance: Analytic models for both designs showed the consequences of FIFO queueing in CORBA and the use of multicast in POSIX.21.

An important aspect of the use of the ATA approach is that one must consider interactions between different architectural attributes. For example, the modifications necessary to have

each display console aware of all tracks were similar in the CORBA and POSIX.21 designs. However, the performance consequences of such a choice have dramatically different results.

In summary, the use of CORBA and POSIX.21 for the resectorization problem highlighted the utility of each design approach. In addition, the use of the ATA approach helped to influence the development of each design.

References

[FAA 95a] Federal Aviation Administration. *National Airspace System Computer Program Functional Specifications: Introduction to Specification Series* (NAS-MD-310). Washington, DC: August 15, 1995.

[FAA 95b] Federal Aviation Administration. *National Airspace System Computer Program Functional Specifications: Message Entry and Checking* (NAS-MD-311). Washington, DC: August 15, 1995.

[FAA 95c] Federal Aviation Administration. *National Airspace System Computer Program Functional Specifications: Local Outputs* (NAS-MD-314). Washington, DC: August 15, 1995.

[FAA 95d] Federal Aviation Administration. *National Airspace System Computer Program Functional Specifications: Adaptation* (NAS-MD-316). Washington, DC: August 15, 1995.

[FAA 95e] Federal Aviation Administration. *National Airspace System Computer Program Functional Specifications: Performance Criteria* (NAS-MD-318). Washington, DC: August 15, 1995.

[FAA 95f] Federal Aviation Administration. *National Airspace System Computer Program Functional Specifications: Software Design Requirements* (NAS-MD-325). Washington, DC: August 15, 1995.

[FAA 95g] Federal Aviation Administration. *National Airspace System Computer Program Functional Specifications: Adaptation Collection Guideline* (NAS-MD-326). Washington, DC: August 15, 1995.

[FAA 95h] Federal Aviation Administration. *National Airspace System, Compool Table Design Specification, Track Control/Display Table* (NASP-5130-HO-P03). Washington, DC: August 1995.

[FAA 96] Federal Aviation Agency. *En Route Architecture Team Study Findings*. Washington, DC: May 1996.

[IEEE 96] Institute for Electrical and Electronics Engineers. IEEE Draft Standard 1003.21, *Information Technology - Portable Operating System Interface (POSIX) - Part 21: Real-time Distributed Systems Communication Application Programming Language Interface (API) [Language Independent]*. Los Alamitos, CA: September 1996.

[IEEE 95] Institute for Electrical and Electronics Engineers. IEEE Emerging Practices in Technology, *Interface Requirements for Real-time Distributed Systems*, IEEE P1003.21 N008 R11. Los Alamitos, CA: July 1995. [Available online: <URL: http://standards.ieee.org/reading/ieee/ept/>].

[Kazman 86] Kazman, R.; Abowd, G.; Bass, L.; and Clements, P. "Scenario-Based Analysis of Software Architecture," IEEE Software, Nov. 1996, 47-55.

[Klein 93] Klein, M; Ralya. T.; Pollak, B.; Obenza, R.; and Gonzales Harbour, M. *A Practitioners Handbook for Real-Time Analysis*. Kluwer Academic, 1993.

[Lipson 97] Lipson, H. and Longstaff, T.(eds.), *Proceedings of the 1997 Information Survivability Workshop*. IEEE Computer Society Press, 1997.

[Meyers 97] Meyers, B. C.; Place, Patrick R.H.; & Kanevsky, A. *A Comparison of CORBA and POSIX.21 From a Real-time Communication Perspective* (CMU/SEI-97-TR-15). Pittsburgh, Pa.: Software Engineering Institute, December 1997.

[OMG 95a] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Version 2.0. Framingham, MA: July 1995.

[OMG 95b] Object Management Group. *CORBA Services: Common Object Services Specification*. Framingham, MA: March 31, 1995.

Appendix A Acronyms

This appendix contains a list of acronyms relevant to this report. The terms come from a variety of domains. In the table below, the second column indicates the domain use of the term in the following manner:

- C: CORBA; from OMG specifications [OMG 95a].
- F: FAA air-traffic control; from FAA study findings [FAA 95a].
- P: POSIX.21; from IEEE standards [IEEE 96].

Term	Use	Meaning
AHI	F	Automatic handoff initiation
ARTCC	F	Air route traffic control center
ARTS	F	Automated radar terminal system
BCP	F	Boundary crossing point
CFAF	F	Central flow automation facility
CID	F	Computer ID
CORBA	C	Common Object Request Broker Architecture
CS	F	Resector message
DRI	F	Distance reference indicator
FDB	F	Flight data block
FPA	F	Fix posting area
HCS	F	Host computer system
ISO	P	International Organization for Standardization
NADIN	F	National Airspace Data Interchange Network
ORB	C	Object request broker
PDA	P	Protocol dependent address

POSIX	P	Portable operating system interface
PS	F	Planned shutdown message
RC	F	Sector assignment request message
SMMC	F	System maintenance and monitor console

Appendix B Glossary

This appendix contains a glossary of terms relevant to this report. The terms come from a variety of domains. In the table below, the second column indicates the domain use of the term in the following manner:

- C: CORBA; from OMG specifications [OMG 95a].
- F: FAA air-traffic control; from FAA study findings [FAA 95a].
- P: POSIX.21; from IEEE standards [IEEE 96].

Term	Use	Definition
Acknowledged data transfer	P	The transmission of data from a source endpoint to one endpoint — or, in the case of multicast, more than one endpoint — and the subsequent response indicating the status of the data transmission.
Activation	C	Preparing an object to execute an operation. For example, copying the persistent form of methods and stored data into an executable address space to allow execution of the methods on the stored data.
Active sector	F	A sector providing air-traffic control in one or more assigned fix posting areas.
Adaptation	F	Data tables that define the unique environment of a given ARTCC. Examples are boundaries, airways, fix posting areas, and input/output devices.
Adaptation data	F	Data available to the operational software. These data include permanent data which define the characteristics of the operating-system environment at a unique location. Examples include geographic data (radar site locations, fix and airway data), aircraft characteristics, design parameters, initial conditions, and other system parameters included in adaptation. Provision is made for modifying adaptation data whenever the real world represented by the stored data changes.
Adapted	F	Contained or preset in adaptation.

Adapted sectorization plan	F	Any one of 10 plans that may be activated by a resector (CS) message.
Adjacent center	F	A center whose area is adjacent to that of the center being discussed.
Airway	F	A named, adapted route defined as a series of adapted fixes and junctions.
Approach control area	F	One or more contiguous fix-posting areas controlled by an approach-control facility. Approach-control airspace may overlie or underlie airspace controlled by ARTCC sectors or adjacent approach-control facilities.
Approach control facility	F	An air-traffic control facility exercising control within a delegated block of airspace.
Asynchronous events	P	Events that occur independently of the execution of an application.
Asynchronous interaction	P	An interaction between schedulable units (processes and/or threads) in which, after a schedulable unit invokes an operation to take part in the interaction, control is allowed to return to the schedulable unit before the operation is completed.
Basic sector plan	F	A sector plan in which each FPA is assigned to the sector identified by the first two digits of the FPA number.
Blocking operation	P	An operation in which, during the execution of the operation, the schedulable unit (process or thread) that invoked the operation cannot continue execution and cannot be rescheduled until some part or all of the operation is completed.
Boundary crossing point	F	The point at a flight's altitude where a boundary crossing between two centers occurs.
Boundary crossing time	F	The time at which a flight is calculated to intersect the boundary crossing point.
Broadcast	P	A non-guaranteed transfer of data from a source endpoint that is available to all other endpoints known to the implementation.
Client	C	The code or process that invokes an operation on an object.
Converted fix	F	A fix developed by the program from the filed route.
Coordination fix	F	A fix used as a common reference point for coordination between facilities.
Connection	P	A logical virtual circuit established between two endpoints.

Crosstell	F	The condition of track handoff between facilities (i.e., ARTS III, NAS). The condition can exist at both sending and receiving facilities.
Current sectorization	F	The arrangement of control sectors and their assigned FPA(s) resulting from the sector plan in effect, plus modification of resectorization (CS) messages.
Data block	F	The symbology displayed adjacent to a tracked or untracked aircraft target on a PVD, containing (subject to field filtering) tracked aircraft or a flight-plan position symbol, leader, velocity vector, and alphanumeric data associated with the aircraft.
Datagram	P	A message transferred between communicating endpoints without the benefit of a connection.
Departure list	F	A list of aircraft to the PVD associated with the sector that is to receive the first flight-progress strip on a departed flight.
Directory	P	A distributed collection of information that applications can access to make queries or updates. The information of interest to communicating applications in a directory includes, but is not limited to, logical names associated with applications and the protocol-dependent addresses at which they are located.
Directory services	P	The set of operations that allows an application to interact with a directory.
Distance reference indicator	F	A 12-sided polygon displayed at a radius of 5 miles about a selected track position.
Endpoint	P	An object created and maintained by the implementation that is used by applications to send and receive data, and by the implementation to identify the source and destination of the data.
Fix	F	Any geographical point.
Fix name	F	An alphanumeric identification of a geographic point that contains 2-12 characters.
Fix-posting area (FPA)	F	A volume of airspace, bounded by a series of connected line segments with altitudes, that is assigned to a sector or approach-control facility. The FPA is the basic unit of space within the air-traffic control system.
Fix time determination (FTD)	F	The establishment and maintenance of stored fix times for each converted fix in each flight plan in the system. This process uses speed and times filed or updated in the flight plan, geographical route and adaptation data, and stored wind data.

Flight data	F	All data applicable to a flight including filed flight plan, flight amendments, reported altitude, track position and velocity, and time estimates.
Flight plan	F	A collection of data relating to a specific aircraft or formation of aircraft, containing all the information necessary for tracking and producing flight-progress strips used to control the flight. The status of a flight plan may be either *proposed* or *active*.
Full data block	F	See *data block*.
Handoff fix	F	A predetermined geographical location over which an aircraft will transit from one area of control to another.
Handoff status	F	The status of a track during the time its control is being transferred from one sector or facility to another (e.g., between initiation and acceptance of handoff).
Implementation inheritance	C	The construction of an implementation by incremental modification of other implementations. The ORB does not provide implementation inheritance. Implementation inheritance may be provided by higher levels.
Inactive sector	F	A sector to which no fix-posting areas are currently assigned.
Inbound list	F	A list of inbound aircraft from adjacent centers or ARTS III facilities within a specified number of minutes of the center boundary crossing for the sector associated with this PVD.
Inheritance	C	The construction of a definition by incremental modification of other definitions. See *interface* and *implementation inheritance*.
Interface	C	A listing of the operations and attributes that an object provides. This includes the operation signatures and attribute types. An interface definition ideally includes the semantics as well. An object *satisfies* an interface if it can be specified as the target object in each potential request described by the interface.
Interface inheritance	C	The construction of an interface by incremental modification of other interfaces. The IDL language provides interface inheritance.
Logical name	P	An identifier known to the application that can be used to refer to an endpoint.

Message	P	A unit of information that can be transferred between communicating endpoints. A message includes application-specific data plus a number of message attributes that may represent information about how the message is controlled (such as priority, message label, and format).
Message label	P	An associated "message type" that is transferred with a message.
Metering	F	A function that provides the capability to determine and display an ordered sequence of aircraft. This function has two major components. One is for metering the arrival of aircraft destined for airports that are external to the center (En Route Spacing Program or ESP). The other is for metering aircraft destined for airports located within the center (Arrival Sequencing Program or ASP).
Method	C	An implementation of an operation. Code that may be executed to perform a requested service. Methods associated with an object may be structured into one or more programs.
Multicast	P	A transfer of data from a source endpoint to a set of specified endpoints.
Multicast group	P	A set of endpoints that are available as a group for communication.
Node	F	The geographical point used to define the horizontal structure of a fix-posting area and/or other lines used in displaying information in a center.
Object	C	A combination of a state and a set of methods that explicitly embodies an abstraction characterized by the behavior of relevant requests. An object is an instance of an implementation and an interface. An object models a real-world entity, and it is implemented as a computational entity that encapsulates a state and operations (internally implemented as data and methods) and responds to requestor services.
Object adapter	C	The ORB component that provides object reference, activation, and state-related services to an object implementation. Different adapters may be provided for different kinds of implementations.
Object reference	C	A value that unambiguously identifies an object. Object references are never reused to identify another object.
Outbound coordination fix	F	The coordination fix transmitted to an approach control or adjacent center.
Persistent object	C	An object that can survive the process that created it. A persistent object exists until it is explicitly destroyed.

Primary FPA	F	The FPA to which another FPA is assigned.
Protocol	P	A set of syntactic and semantic rules for exchanging information.
Reconfiguration	F	*(1) Automatic reconfiguration:* an action taken by the operational software to recognize a failure and switch the failed element or devices out of the operational system and replace it with a standby unit. (2) *Manual reconfiguration:* Similar to automatic reconfiguration except that it is caused by an input from a supervisory position.
Resectorization	F	The act of changing the FPAs and sectors assigned to the various sectors according to one of the sectorization plans.
Route (of flight)	F	A defined path, consisting of one or more route segments that an aircraft traverses over the surface of the earth.
Route segment	F	A part of a flight route, consisting of two fixes and the route between them.
Sector	F	A geographical area limited to an altitude within an ARTCC that may contain one or more related control positions.
Sector airspace	F	One or more contiguous fix-posting areas controlled from a single control sector (i.e., the FPAs assigned to a sector). A sector's airspace may overlie or underlie airspace controlled by another sector or by an approach-control facility.
Sector plan	F	An adapted set of sector/FPA assignments that may be implemented by reference to a unique plan name. (See also *sector plan*)
Server	C	A process implementing one or more operations on one or more objects.
Signature	C	Defines the parameters of a given operation including their number and order, data types, and passing mode; the results, if any, and the possible outcomes (normal vs. exceptional) that might occur.
Subjugate FPA	F	An FPA that is assigned to a primary FPA by means of adaptation. (See also *Primary FPA*)
Synchronous interaction	P	An interaction between schedulable units in which, after a schedulable unit invokes an operation to take part in the interaction, the schedulable unit is blocked until the operation is completed.
Track	F	The computer-generated representation of an aircraft's position and movement.

Transient object	C	An object whose existence is limited by the lifetime of the process or thread that created it.
Unicast	P	A transfer of data from a source endpoint to a destination endpoint.

Appendix C Possible Sector Changes

It is appropriate to provide some discussion of the operations on FPAs and sectors that are currently not supported. These possible operations may be worth considering for a future enhancement of the system.

C.1 Taxonomy of Operations

A number of possible operations on an FPA or sector are possible. It is useful to develop a taxonomy of operations. We shall therefore consider the following classes of operations:

- operations on a node
- operations on a line segment
- operations on the FPA as an entity

Operations on a node view an FPA *as a set of independent nodes*. In this case, because of the set property, an operation on a node is independent (explicitly) of any other node. There are, however, implicit dependencies that are possible. For example, if a node is deleted from an FPA, there must be a procedure to ensure that the result of the operation leaves the FPA a closed object.

Operations on a line segment are based on the view that an FPA is a *set of line segments*. Again, any operation on a line segment must include a procedure to ensure that the resulting FPA maintains a closed boundary. These operations are implicit.

Operations on an FPA as an entity are performed on all elements of the FPA. Depending upon the operation, it may be more appropriate to perform the operation on the nodes or the line segments. Examples of operations that fall into this class include scaling, translating, and rotating an FPA.

C.2 Operations on One Node

The first set of operations to be discussed are those that relate to an individual node. These are relatively simple to present. The first case to be considered is that of moving an existing node. This option is presented in Figure 84 below. The notation of Figure 84 will be used throughout this section and is as follows:

- The left side of the figure denotes the initial state, while the right side denotes the result of an operation.
- An open circle denotes a node of an FPA or sector. An open circle on the right side of a figure means that the node is unchanged. If the node will be affected by the operation, it appears as a filled circle on the left side. If the node is filled on the right side, it means that the node changed.
- The FPA or sector boundary is presented as a solid line on the left side of a figure, but as a broader, shaded gray line on the right side of the figure if the corresponding boundary changes as a result of the operation.
- The lines from the FPA or sector boundary that connect to another FPA and/or sector are indicated by a solid line on the left side of a figure. If, as a result of an operation, the lines change, they appear as dashed lines on the right side of the figure.

Note that when a node is moved to another location, as indicated in Figure 84, it causes the following additional changes:

- Two line segments of the FPA must be recomputed.
- The boundary of the FPA must be recomputed.
- The line segments and FPA boundary of the adjacent FPA must be recomputed.

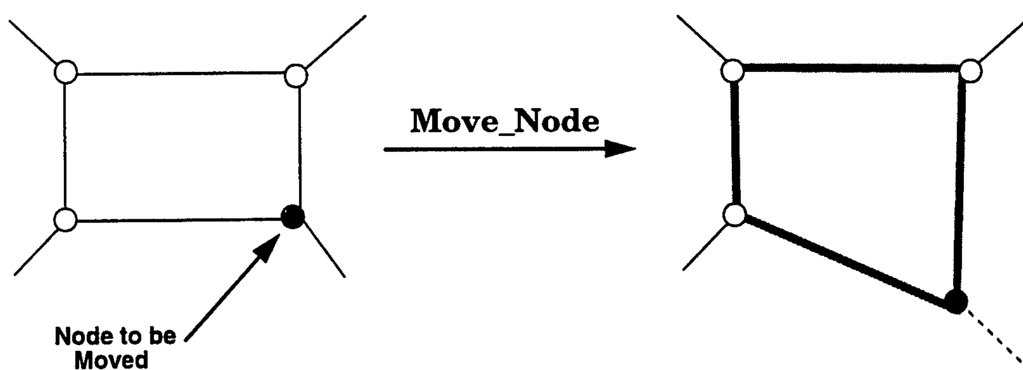


Figure 84: Moving an Existing FPA Node

The second case is that of adding a new node to an FPA. This case is shown in Figure 85 below. This results in the creation of two line segments and the deletion of a previously defined line segment. It will also change the structure of the adjacent FPAs.

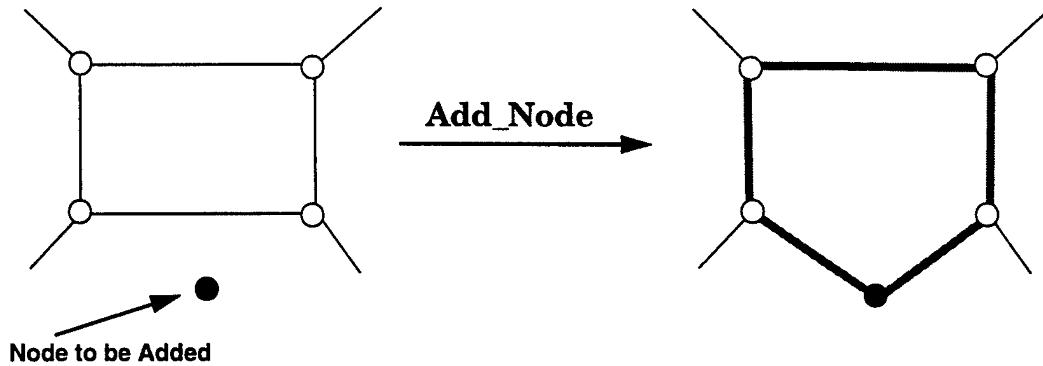


Figure 85: Adding a Node to an FPA

The last case of a node-specific operation to consider is that where an existing node is deleted. This is shown in Figure 86 below. Deleting an existing node results in a decreased number of line segments in the FPA. There are also a number of options for what happens to the line segments of any FPAs that were adjacent to the FPA that has a node deleted. For example, the deleted node could be extended toward the new line segment; this possibility is shown in the dashed line in the figure. After a choice has been made for the "dangling line segment," the remaining line segments of the adjacent FPAs would need to be recomputed.

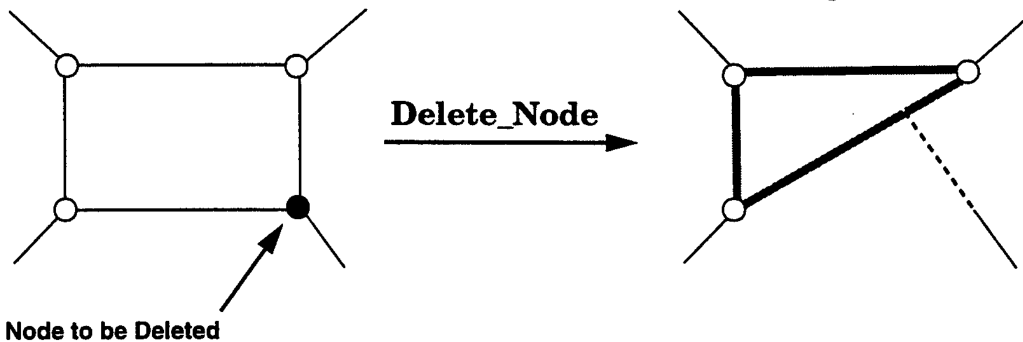


Figure 86: Deleting a Node from an FPA

C.3 Operations on Multiple Nodes

It is also possible to envision operations that are simultaneously performed on an FPA's set of nodes. One opportunity here is where two (or more) nodes would be coalesced to either an existing node or a new node. An example of this is shown in Figure 87. The point indicated

with ✚ will be called a *coalescent point*; this means that the nodes that are to be coalesced will be moved to this point. This type of operation is clearly more complex than an operation on an individual node. For example, if two nodes as coalesced, as indicated in the figure, a new node must be computed (the coalescent point), and two new line segments of the FPA must be recomputed. Furthermore, additional computation of the line segments of the adjacent FPA(s) must be recomputed. Note that in Figure 87 we have assumed that the line segments of the adjacent FPA will be connected to the coalescent point. This seems to be a reasonable assumption.

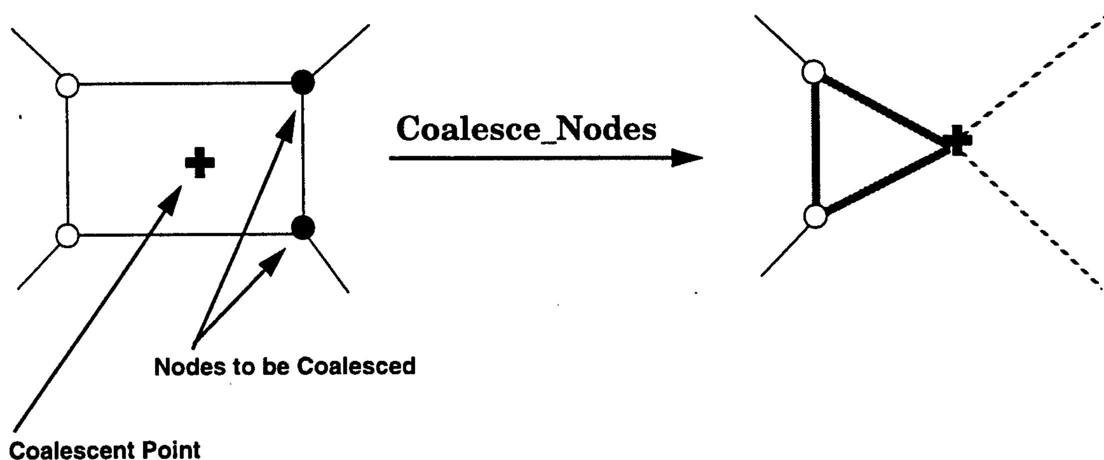


Figure 87: Coalescing Two Nodes Into One

C.4 Operations on Line Segments

Another class of operations are those that are performed on a line segment. The first type of operation is that in which a new line segment is created in an FPA. This case is illustrated in Figure 88, where the dashed line is the line segment to be added. The result of adding the line segment results in a *partition* of the original FPA, as indicated on the right side of Figure 88. If this type of operation were supported, it would be necessary to define a new FPA. However, note that the nodes in this case are invariant with respect to the creation of the new line segment.

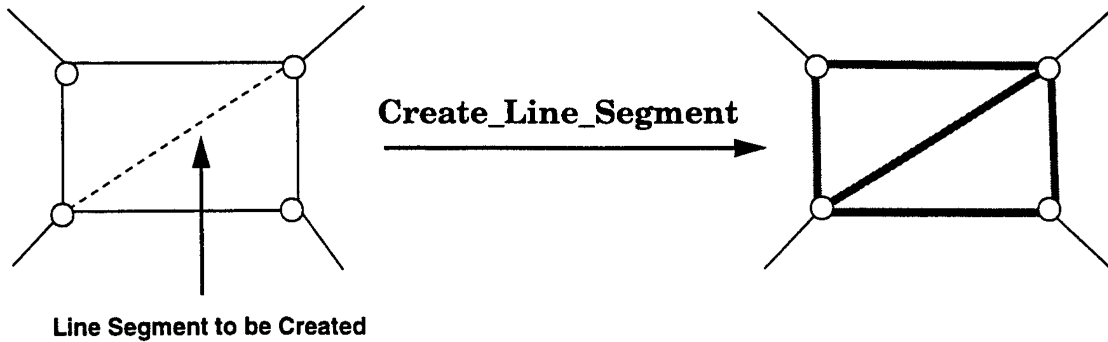


Figure 88: Creating a New Line Segment

A second type of operation on a line segment is where a line segment of an existing FPA is moved. An example of this is illustrated in Figure 89 below. Here, the line segment on the right has been *translated* to the right, resulting in a change to the FPA. An additional consequence of this operation is that the line segments of the adjacent FPA must be recomputed since two nodes have changed.

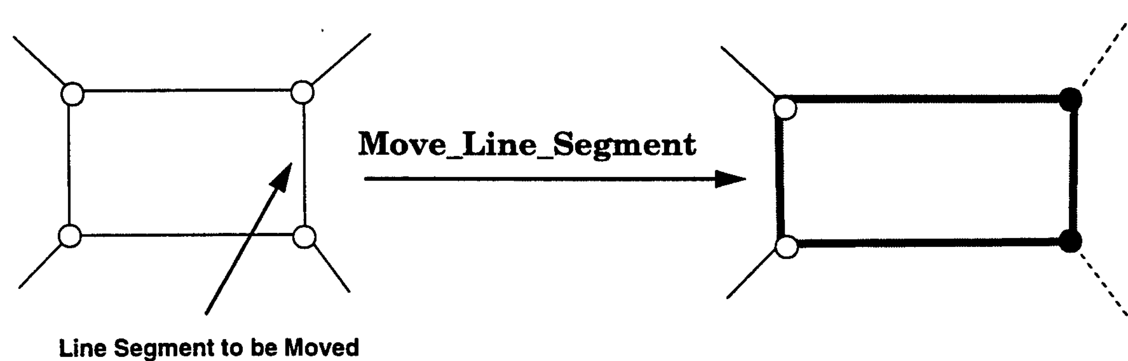


Figure 89: Moving a Line Segment

The next type of operation to consider on a line segment is where a line segment is deleted. An example of this is shown in Figure 90. When a line segment is deleted, some choice must be made to maintain connectivity of the FPA. The choice made in Figure 90 is to use the midpoint of the deleted segment to implicitly create a new node, which is then connected to the appropriate line segments.

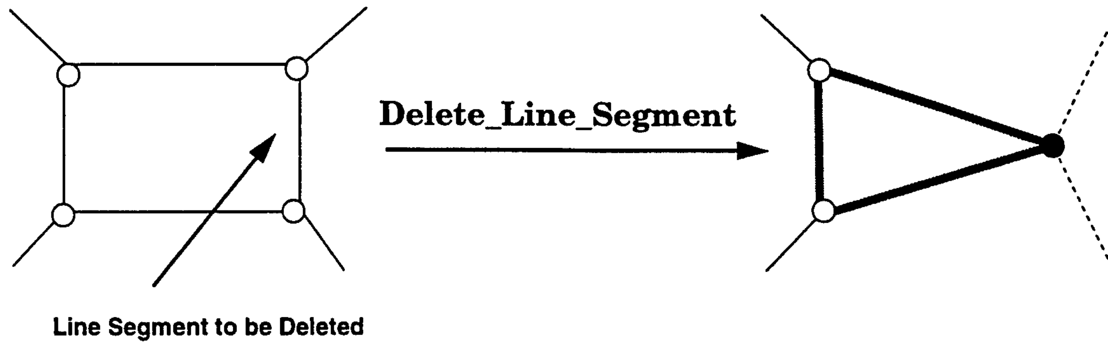


Figure 90: Deleting a Line Segment

Another possibility is where an FPA is translated a specified distance in a specified direction. This is illustrated in Figure 91 where the FPA has been translated to the right. Note that under the translation, all the nodes must be recomputed, as well as the line segments that connect the nodes to nodes in other, adjacent FPAs. The changes in the line segments that connect to an adjacent FPA are shown as dashed lines in Figure 91.

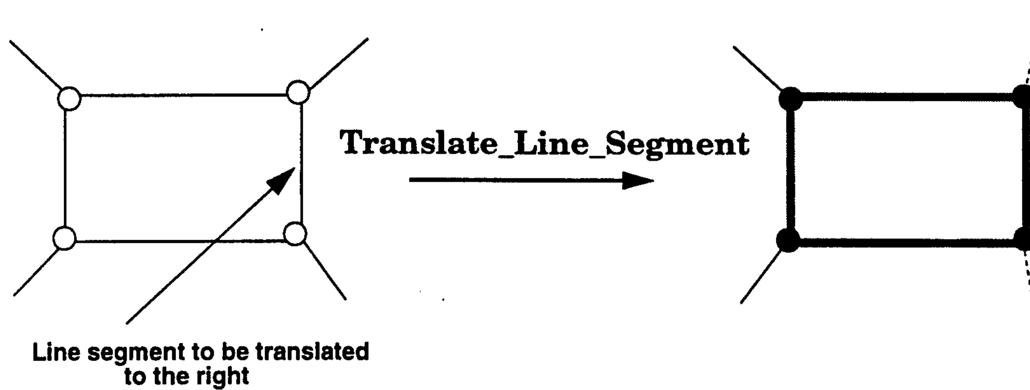


Figure 91: Translation of a Line Segment

More complex operations on a line segment can also be considered. For example, Figure 92 shows the result of translating and rotating a line segment that forms the boundary of an FPA. As before, the nodes of the FPA would need to be recomputed, as well as the line segments of any adjacent FPAs.

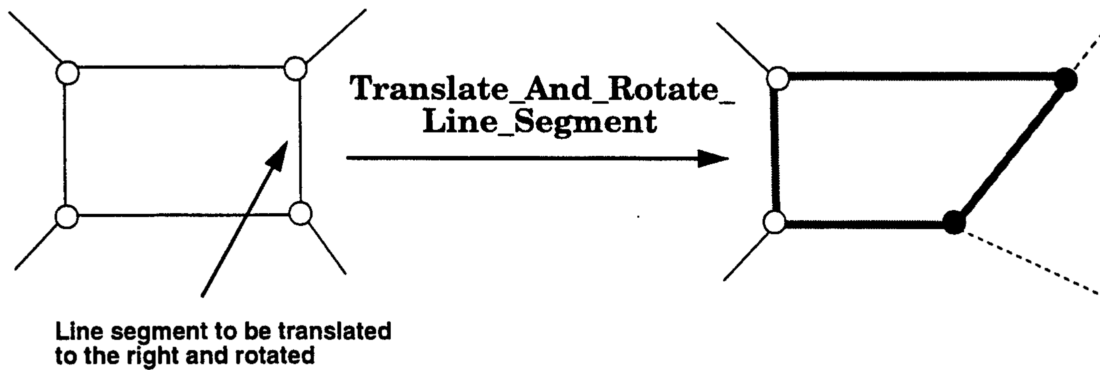


Figure 92: Translation and Rotation of a Line Segment

C.5 Operations on a Geometric Entity

The final class of operations to discuss are those that are performed on an FPA, treated as an entity. The first case is that where an FPA is scaled. This is shown in Figure 93 where a scale factor greater than unity is assumed.

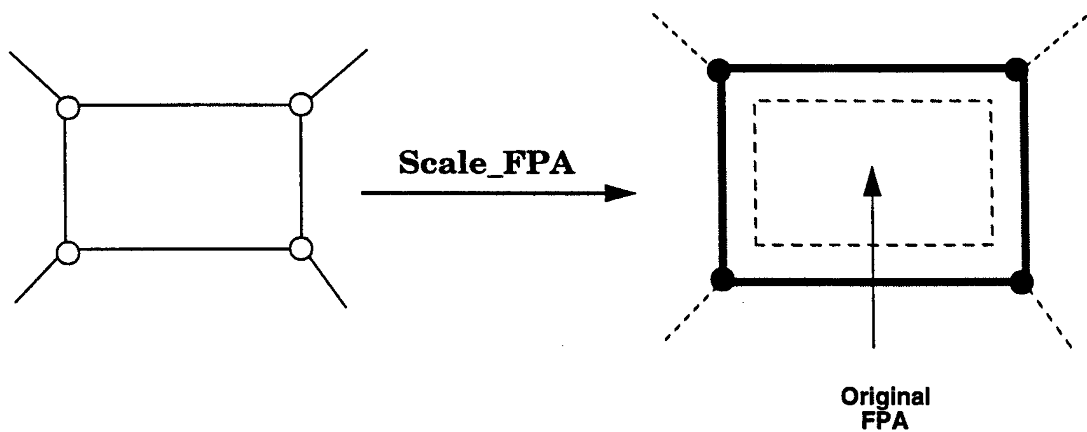


Figure 93: Scaling an FPA

Another possibility is that where an FPA is translated a specified distance in a specified direction. This is illustrated in Figure 94 below where the FPA has been translated toward the right. Note that under the translation, all the nodes must be recomputed, as well as the line segments

that connect the nodes to nodes in other, adjacent, FPAs. These are indicated by dashed lines on the right side of the figure.

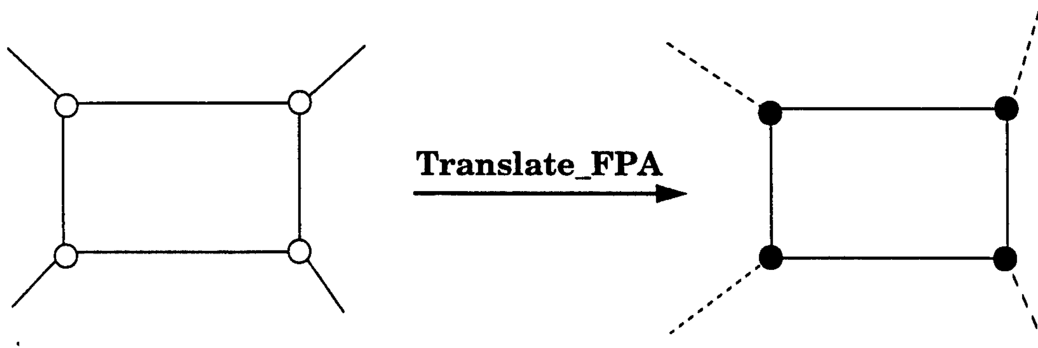


Figure 94: Translating an FPA

The final type of operation that we will consider on an entire FPA is that where an FPA is rotated. This is illustrated in Figure 95, where the indicated FPA has been rotated in the clockwise direction by about 30 degrees. As a result of the rotation, it is necessary to determine the locations of the new nodes, as well as the line segments from adjacent FPAs that connect to these nodes. These are shown as dashed lines on the right side of the Figure 95.

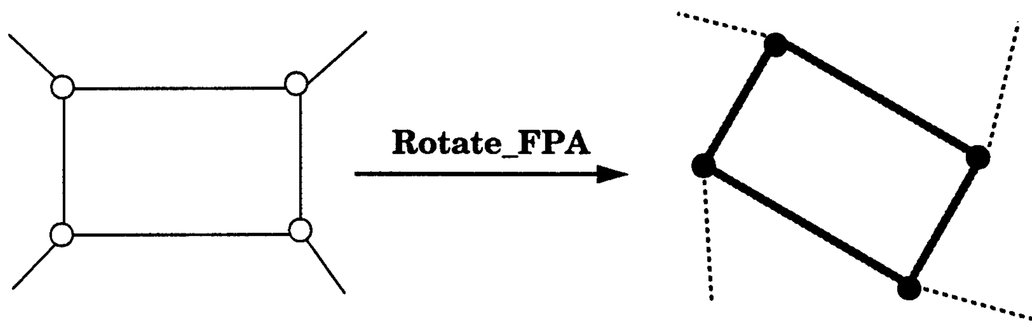


Figure 95: Rotating an FPA

Another obvious case is that where an FPA undergoes both a translation and a rotation.

C.6 Summary

Permitting dynamic sector and/or FPA boundaries is an interesting concept, particularly in the context of free flight. In many cases, this would require recomputing node positions (which are defined in adaptation data) and ensuring that certain constraints on node separation are assured. Currently, the constraints on node separation are performed off-line. However, allowing dynamic operations, such as those discussed above, would require that this processing be performed during runtime. There are also additional constraints that would need to be determined in order to accept a particular dynamic change to an FPA structure.

It is important to remember that in the current specification of the En Route system, all FPAs are *indivisible* geometric objects. Moreover, sectors are combinations of one or more FPAs. The operations discussed above would relax this fundamental assumption, by allowing, for example, an FPA to become a dynamic object.

Appendix D Additional Requirements Specification

D.1 Adaptation Data Management

Adaptation data are those data that are unique to a given ARTCC and permit a certain tailoring of displays, and so forth. Different types of adaptation data shall be supported including

- radar-console display configuration
- sector plan adaptation data
- FPA adaptation data
- node adaptation data

D.1.1 Radar-Console Display Configuration

Each radar display console shall have a coordinate system in an xy-plane, where both x and y values are in the range 0 to 1023, inclusive.

The capability shall be provided to specify the locations of various data elements that are displayed on a radar display console.[1] The following data elements may be specified in adaptation data for a specified console:

- UTC time display
- hold list display
- inbound list display
- departure list display
- conflict list display

1. The following is based on a discussion of the *Plan View Display Record* on pages 7-22 through 7-31 of FAA specifications [FAA 95g]. Additional elements, listed in the indicated pages, are omitted.

- conflict-alert status display
- VFR inhibit display
- sector-meter list display

In addition, the capability shall be provided to specify the above data elements such that they are applicable to all radar display consoles.

The format of the above data elements is defined in FAA specifications [FAA 95g].

D.1.2 Management of Sector-Adaptation Data

Sector adaptation data refer to that set of data that is used to describe the adaptation characteristics of a sector. This includes the following types of data:

- sector configuration, defining the devices associated with a sector, and certain attributes of a sector, such as whether or not the sector is permanently defined as a training sector
- sector plans, which define one or more possible sector configurations of an En Route center. Sector plans are used to initialize and/or reconfigure a facility

D.1.2.1 Management of Sector-Configuration Adaptation Data

Sector configuration adaptation management refers to the capability to assign one or more local devices, and several constants, to a particular sector as part of system initialization. For each sector number, it shall be possible to specify the following parameters:[1]

- a sector name
- the logical address of the flight-strip printer associated with this sector
- D-position address: specifies both the logical and physical address of the D-position display device
- R-position address: the (physical) address of the R-display
- data-block offset; indicates a set of values that denote where the track-data tag should be displayed
- training indicator, indicating that the sector is permanently assigned to be a training sector
- sector conflict-alert boundary constant

---

1. The following text is based on pages 5-20 through 5-24 of FAA specifications [FAA 95g]. It is interesting to note that some of the items described in the indicated reference are required to be present, some are optional, and some are neither optional nor required.

- sector conflict-alert MCI boundary constant
- sector altitude type; this parameter may assume values indicating that the sector is one of the following:
  - low-altitude sector
  - high-altitude sector
  - adjacent to an ARTS facility

The following requirements apply to the overall specification of a facility sector configuration, defined as the union of all individual sector plans:

- Each sector name and sector number shall be unique.
- More than one sector may be assigned to the same logical name of a flight-strip printer.
- The set of logical names of flight-strip printers shall be disjointed from the set of logical names for D-consoles.
- Each physical address shall be unique.
- A capability shall be provided to define multiple values of the parameters for device specification (i.e., flight-strip printer, R-console, and D-console) in a sector configuration to be used for backup purposes. The specification of such backup parameters is subject to the first four items above.

D.1.2.2 Management of Sector-Plan Adaptation Data

A sector plan defines a set of sectors and/or FPAs that encompass the area of an En Route center. Sector plans, which include information about sector topology and associated map data, are defined in adaptation data in order to provide the versatility to allow a given center to be configured to meet its unique needs. The maximum number of sector plans shall be 10.

D.1.2.3 Basic Sector Plan

The *basic sector plan* is a default sector plan in which there is a predefined relationship between a sector and one (or more) FPAs. Only one basic sector plan shall be permitted.[1] The format of a basic sector plan shall be as defined in FAA specifications [FAA 95g].

D.1.2.4 Derived Sector Plans

1. The basic sector plan is denoted as plan 0, although this sounds like implementation.

A *derived sector plan* is a sector plan that is defined as a modification of the basic sector plan. The capability shall be provided to define one or more derived sector plans subject to the following requirements:

- The maximum number of derived sector plans shall be less than or equal to nine.
- Only the effected sectors and/or FPAs shall be required to be specified in a derived sector plan; all other sectors and/or FPAs remain as defined in the basic sector plan.
- A derived sector plan may specify that one (or more) sectors and/or FPAs be attached to a specified sector. When a sector and/or FPA is attached to a specified sector, all associated postings, communication, and so forth, shall be associated with the specified (receiving) sector.
- FPAs shall not be combined in a derived sector plan.
- The format of a derived sector plan shall be as defined in FAA specifications [FAA 95g].[1]

D.1.3 Management of Fix-Posting-Area Adaptation Data

Each FPA shall be denoted as *<sector_id>* *<fpa_id>* where both *<sector_id>* and *<fpa_id>* shall be in the range from 0 to 99, inclusive.

D.1.4 Input Message Eligibility Record

The system shall provide a mechanism to configure a console with respect to the requests that may be initiated from a particular console. This information shall be provided in the input message eligibility record. The format of this information shall be as specified in FAA specifications [FAA 95g].[2]

D.1.5 Node-Adaptation Data Management

A node is a geographic point that is used to define the horizontal structure of an FPA, B-line, S-line, and/or transition line. The following requirements apply to node-adaptation data management:

- A node shall be specified by an identifier of two to four alphanumeric characters.
- All node identifiers shall be unique within the context of a specific En Route Center.

1. In particular, see pages 5-24 through 5-26 of FAA specifications [FAA 95g].
2. In particular, see Section 5.3 of the referenced document.

- A node shall be specified in terms of latitude and longitude, expressed in degrees, minutes, and seconds.

The following restrictions apply to those nodes that are used to define an FPA:

- Each FPA node shall be at least one-quarter of a mile from any other node of that FPA.
- Each FPA node shall be at least one-quarter of a mile from any line of that FPA that does not contain the node.
- Each FPA node must be at least one-quarter of a mile from a different node of another FPA.

Appendix E Message Contents

A number of messages are currently used to convey information about flight plans or track data. The following is a brief summary of the contents of the messages referenced in the text, listed in alphabetical order. The information is taken from FAA specifications [FAA 95b] which may be consulted for detailed information about messages.

Activate Flight Plan: message type, flight plan ID

Amend Flight Plan: message type, flight plan ID, field reference identification, field reference location

Assign Flight Plan Altitude: message type, flight plan ID, logic check override, assigned altitude

Assign Flight Plan Beacon Code: message type, flight plan ID, logic check override, beacon code

Deactivate Flight Plan: message type, flight plan ID

Delete Flight Plan: message type, flight plan ID

Drop Track: message type, logic check override, flight identification

Flight Plan: message type, flight plan ID, aircraft data, beacon code, speed, coordination fix, coordination time, requested altitude, route information, comment text

Hold Flight Plan: message type, flight plan ID, hold data

Track: message type, flight identification, speed, assigned altitude, heading, logic check override, action type, trackball coordinates, primary track class indicator

Update Flight Plan: message type, flight plan ID, aircraft data, beacon code, speed, coordination fix, coordination time, requested altitude, route information, comment text

Appendix F Specification of Loading Conditions

For the development of the mathematical models presented in this report, it is necessary to determine the load that is placed on a system element. For example, the maximum number of track and flight-plan messages must be determined. One reason for this is network bandwidth considerations. Another reason is because the maximum loading defines the size of queues that can exist in a component. Queue size is a special concern for display consoles because the queueing of track messages, for example, can delay the time it takes for resectorization processing to complete.

The basic model we shall used is based on Appendix A of FAA specifications [FAA 95e]. The basic assumptions are as follows:

- 1100 active flight plans during one hour
- 700 tracks

Given the rate of messages per flight, as indicated in FAA specifications [FAA 95e], we may estimate the total number of information transfers per hour. Then, we assume there are a total of 50 radar display consoles, each of which is equally loaded. Based on this information, the load is summarized in Table 20.

The track update rate for the En Route center assumes a value of one track update per six seconds. If 700 tracks are equally distributed over 50 consoles, this implies 14 tracks per console being updated every 6 seconds. The dominance of track data is clear.

Note that the above assumptions do *not* represent a worst-case loading condition. A number of message transfers have not been included, such as other messages from external En Route centers (although the fact that some are included, such as initiate handoff, weights the numbers to those sectors that are adjacent to an external En Route facility), conflict-alert messages, and other possible system operator commands. Nor does the above account for information transfers that could be initiated by a radar console operator. It also does not account for any information transfer to support the underlying network infrastructure, such as possible routing requests, heartbeats, and so on. Any of these cases can lead to additional preemption and

blocking with respect to processing a resectorization command. However, to account for this level of detail, it would require a much more thorough analysis than in the current scope.

Information	**Total/ hour**	**Total/ console/ hour**
Flight plan message	1,100	22
Flight plan departure message	550	11
Flight plan drop message	220	4
Flight plan altitude amendment	770	15
Flight plan route amendment	660	13
Initiate handoff	2,090	42
Accept handoff	2,530	50
Initiate transfer	1,100	22
Accept transfer	1,100	22
Track update (from external En Route center)	11,600	232
Track update (from this En Route center)	660,000	13,200

Table 20: Assumptions About Console Loading

Appendix G Details of CORBA Approach

G.1 Description of Objects

We present here some information on how objects could be described. The information could include

- object name
- state data associated with the object
- methods available on the object

For the methods available, associated information might contain the method name, description, parameters, and information about the implementation, such as the entities that invoke the method.

For the purpose of this work, we will concentrate on the state data for an object and the name of the method, along with a brief description. Details about parameters are viewed as being appropriate for a lower level of design activity.

G.2 Sample IDL Description

In this section we will present the Interface Description Language (IDL) for a typical CORBA example. This is intended to provide a high-level view of what the IDL would look like. Thus, the following specifies the IDL for a track object.

```
module Track_Obj {

typedef string Obj_Ref;          // Object Reference type
struct Valid_Time {                                          // Valid Time Type
   unsigned long Update_Time_Sec;    // Time in seconds
   unsigned long Update_Time_uSec;   // Time in microseconds
  };

exception Invalid_Time { };
```

```
struct Track_Data {
  double      CID;           // Computer Identifier
  double      X_Position;    // X Position of track in Nautical Miles
  double      Y_Position;    // Y Position of track in Nautical Miles
  double      Altitude;      // Altitude track in Feet
  double      Speed;         // Speed of track in Knots
  double      Heading;       // Heading of track in Degrees
  double      Beacon_Code;   // Track Beacon Code
  Obj_Ref     Flight_Plan;   // Object reference of flight plan
  Valid_Time  Time;          // Valid time of data
};

interface Track {
   // Update Track Parameters
   void     Update_Parameters(in Track_Data update);
                  // Return Track Parameters
   Track_Data Get_Parameters();
              // Extrapolate Track
   Track_Data Extrapolate(in  Valid_Time t) raises (Invalid_Time);
};

interface Factory {
  // Create and return an instance of a track object
  Track CreateTrack(in Track_Data initial);
};

};
```