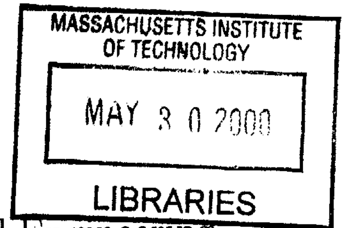
Architecture for the Integration of Dynamic Traffic Management Systems

ENG



by

Bruno Miguel Fernández Ruiz

B.S. Universidad Politécnica de Madrid (1997)

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of
Master of Science in Transportation
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

© Massachusetts Institute of Technology 2000. All rights reserved.

Author ....
Department of Civil and Environmental Engineering
March 31, 2000

Certified by ....
Moshe E. Ben-Akiva
Edmund K.Turner Professor of Civil and Environmental Engineering
Thesis Supervisor

Certified by ....
Didier Burton
Research Affiliate, Center for Transportation Studies
Thesis Supervisor

Certified by ....
Haris N. Koutsopoulos
Operations Research Analyst, Volpe National Transportation System Center
Thesis Supervisor

Accepted by ....
Daniele Veneziano
Chairman, Department Committee on Graduate Students

Architecture for the Integration of Dynamic Traffic Management Systems

Bruno M. Fernandez Ruiz

Submitted to the Department of Civil and Environmental Engineering
on March 31, 2000
in partial fulfillment of the requirements for the degree of
Master of Science in Transportation

Abstract

Dynamic Traffic Management Systems (DTMS) are intended to operate within Traffic Management Centers (TMC) to provide pro-active route guidance and traffic control support. The integration of multiple DTMS software systems requires the modification of the structure and design of the TMCs where they will be integrated. An open, scalable and parallel system architecture that allows the integration of multiple DTMS servers at minimum development cost is presented in the current research. The core of the architecture provides: a generic distribution mechanism that extends the Common Object Request Broker Architecture (CORBA); a generic creation mechanism based on the Abstract Factory pattern that permits an anonymous use of any DTMS within TMCs; and a generic naming mechanism (Registry) that allows the TMC to locate the DTMS servers in remote hosts without using any vendor specific mechanism. Finally, the architecture implements a Publisher/Subscriber pattern to provide parallel programming on top of the CORBA's basic synchronous communication paradigm.

This system architecture is used to propose TMC application designs. The system architecture was validated in a case study that showed the integration of DynaMIT, a prediction-based real-time route guidance system with MITSIMLab, a laboratory for the evaluation of Dynamic Traffic Management Systems. MITSIMLab includes a Traffic Management Simulator (TMS) that emulates the TMC operations. DynaMIT was integrated within TMS using the proposed system architecture.

The core of the system architecture was distributed under CORBA using IONA Technologies Orbix 2.0 Object Request Broker, and it was implemented in C++ using the object-oriented paradigm.

Thesis Supervisor: **Moshe E. Ben-Akiva**
Professor of Civil and Environmental Engineering
Massachusetts Institute of Technology

Thesis Supervisor: **Didier Burton**
Research Affiliate, Center for Transportation Studies
Massachusetts Institute of Technology

Thesis Supervisor: **Haris N. Koutsopoulos**
Operations Research Analyst
Volpe National Transportation System Center

Acknowledgments

I am really thankful to Professor Moshe Ben-Akiva for his guidance, support, and insights through my graduate study. My heartfelt gratitude also goes to Dr. Didier Burton and Dr. Haris Koutsopoulos for leading me the way along my research.

I would like to express my gratitude to the Oak Ridge National Laboratories, the Federal Highway Administration for their financial support. My thanks also go to MIT's Department of Civil and Environmental Engineering and the Center for Transportation Studies for their graduate program.

I consider myself very fortunate for having pursued my graduate study at MIT. My experience at MIT however goes well beyond education. If I had to tell which is the most important lesson I have learned here, getting to know new people would definitely be it. Along the last two years I have developed very intense friendships with outstanding people to whom I will always be in debt. Each of them gave me hints, and made me look at different ways of understanding life. These friendships will last forever and the experience will lead our paths through life. Among these friends are Tomer, Masroor, and Mathew, and all my other fellows from the ITS Program; and Hilton, Wendy, Tatsuo, Maria-Carla, Peter, Coralie, Markus, and Michael, and all others from my beloved Wednesday dinners.

Of course, this journey would not have been possible without the love, time and support of Inge, my girlfriend, to whom I will always be grateful.

Finally, my thanks to my parents for giving me the chance to receive a good education and, most of all, for making me feel loved all the time. They have given me the greatest of all gifts: amor.

Contents

1 Introduction 11
1.1 Dynamic Traffic Management Systems 11
1.2 TMC overview 12
1.2.1 Surveillance Systems 14
1.2.2 Control Systems 14
1.2.3 Incident Management Systems 15
1.2.4 Decision Support Systems 15
1.2.5 Other Systems 16
1.2.6 Computing Environments 16
1.3 Research Objective 17
1.4 Literature review 19
1.4.1 Dynamic Traffic Control 20
1.4.2 Prediction-based (Dynamic Traffic Assignment) ATIS 21
1.4.3 Dynamic Traffic Management Systems 23
1.4.4 TMC architectures 24
1.5 Thesis outline 26

2 Requirements 28
2.1 TMC subsystems description 28
2.1.1 Communication Layers 30
2.1.2 Logic Layers 31
2.2 Architectural requirements 33

3 System Architecture 38
3.1 Distributed Object Computing 38
3.2 Distributed Systems Technology 39
3.2.1 Sockets 40
3.2.2 HTTP/CGI 40
3.2.3 CORBA 41
3.2.4 DCOM 42
3.2.5 RMI 43
3.2.6 Conclusion 44
3.3 CORBA overview 44
3.4 CORBA extensions 48
3.4.1 Inter-operability 49
3.4.2 Concurrent programming 56
3.5 System distribution 64
3.5.1 Four-layer client/server distribution 65
3.5.2 Three-tier distribution 67
3.5.3 TMC system architecture 71

4 Application design 76
4.1 Integrated TMC system design 76
4.1.1 Centralized system, single region 78
4.1.2 Decentralized system, multiple regions 79
4.2 Other applications of the proposed architecture 80
4.3 Conclusion 84

5 Case Study 85
5.1 Requirements 85
5.2 MITSIMLab 86
5.3 DynaMIT 90
5.4 Implementation of the integration of DynaMIT in MITSIMLab 94

6 Conclusions **100**

6.1 Research Contribution 100

6.2 Future Work 101

A UML Object diagram **107**

B Object definition files **108**

C Message definition files **123**

List of Figures

1-1 Generic Traffic Management Node 23

1-2 Iterative prediction/control/guidance procedure 25

2-1 Connection and Logic Layers of a TMC 29

3-1 A request being sent through the Object Request Broker 44

3-2 The structure of Object Request Broker interfaces 45

3-3 A Client using the Stub or Dynamic Invocation Interface 46

3-4 An Object Implementation Receiving a Request 47

3-5 Interface and Implementation Repositories 48

3-6 The Registry pattern 53

3-7 The Factory pattern 56

3-8 The Source/Listener pattern 60

3-9 The Source/Channel/Listener pattern 62

3-10 Four-layer distribution architecture 66

3-11 Client/server distribution architecture 68

3-12 Three tier distribution architecture 70

3-13 Integrated TMC distribution architecture 73

4-1 TMC design: connection of two systems 77

4-2 TMC design: centralized system, single region 78

4-3 TMC design: decentralized system, multiple regions 79

4-4 TMC design: static system with ATMS/ATIS 81

4-5 TMC design: off-line system for DTMS evaluation 82

4-6 Multiregional simulation 83

5-1 Structure of MITIMSLab 87
5-2 TMS: proactive traffic control and routing systems 89
5-3 Structure of DynaMIT 91
5-4 DynaMIT: Illustration of the rolling horizon 92
5-5 Integration of DynaMIT into MITSIMLab: system design 95

Chapter 1

Introduction

In the last decade, research in Intelligent Transportation Systems (ITS) has led to the development of various tools for the optimization of transportation systems. ITS has concentrated some of its efforts on the study of road traffic. Currently, several ITS technologies intended to achieve efficiency in the management of traffic operations are being developed. These technologies are based on complex software systems implemented in the Traffic Management Centers (TMCs). Advanced Traffic Management Systems (ATMSs) and Advanced Traveler Information Systems (ATISs) are the two main ITS technologies currently being developed. The use of both technologies in TMCs will lead to advanced traffic management with dynamic route guidance and traffic control in the future. However, the introduction of such diverse software systems poses difficult problems for issues of system integration, data communication, interfacing , and synchronization, among others. The study of these difficulties and the possible solutions is the basis of this work.

1.1 Dynamic Traffic Management Systems

Dynamic Traffic Management Systems (DTMSs) are designed to support the operations of TMCs. They are the latest generation of support systems and all internal operations are generally performed in a completely automated way. DTMS are dynamic: the management of the traffic network is based on proactive strategies

as opposed to reactive strategies. Predicted conditions constitute the basis upon which these proactive systems generate strategies. Predicted conditions are used to generate route guidance and traffic control.

There is a very strong coupling between traffic control and route guidance, both in the modeling level as well as the physical level. The modeling issues are partially reviewed in 1.4. The physical issues include communication networks, system interfaces, database management, etc.

DTMSs may have varying levels of automation. At the most automated level, all operations (data collection, data processing, decision support, and data dissemination and execution) are performed by a group of software elements. DTMS must interact in real-time among themselves and with the other systems in the TMC. This requires the TMC to provide an open communication architecture that allows the integration of all core systems and DTMSs.

1.2 TMC overview

A TMC consists of multiple ITS systems: some core systems (e.g. the *Surveillance System*), and some interfaced systems (e.g. a Route Guidance System). The core systems provide the basic set of functionality needed to operate a TMC (collecting surveillance data, controlling signals, and coordinating incident response). The core systems are usually *legacy*[1] systems designed for a custom TMC. The TMC basic functionality is enhanced with the addition of interfaced systems (e.g. real-time adaptive traffic control, route guidance, etc.) The TMC must provide the necessary interfacing capabilities. Namely, the TMC must provide:

- A system architecture that allows the integration of all systems. The architecture must encapsulate the custom nature of the TMC's core systems, so that any additional system can be easily plugged in.

[1]In the IT industry, the term *legacy system* is used to design an antique system, normally that does not have public interfacing capabilities.

- Interfaces to other ITS components, such as Advanced Traveler Information Systems (ATIS).
- Interfaces to non-ITS components, such as police, fire and local organization.
- A unified database structure to support the integration, operating efficiency, and interface to other systems.

The fact that different TMCs may have different number of core systems suggests that the design of a generic TMC system architecture that allows the integration of any DTMS should provide encapsulation mechanisms to hide the custom nature of the core systems. These encapsulation mechanisms are provided through a series of interfaces. To obtain the maximum flexibility out of the system architecture, every system should have an interface.

A *thin interface* locates all the functionality in the subsystems, leaving the clients with a minimum shell, most of the times a Graphical User Interface. This provides freedom and expandability to the system interaction, because the functionality is finely distributed among many small systems. A *fat interface* locates more functionality in the clients and somehow reliefs the load of the server systems. This limits expandability, but it is easier and less risky to develop. The ideal interface will be somewhere in between: it is determined by finding the minimum functionality that the system must provide to the TMC, and then finding a compromise in terms of development risk and system expandability.

In order to design the required interfaces it is therefore important to identify and describe the functionality of the subsystems the TMC may consist of.

FHWA (1993) reviewed the state-of-the-practice of several TMCs and identified some of the desirable objectives:

- Collection of real-time traffic data and area-wide surveillance and detection.
- Integrated management of various functions including demand management, toll collection, signal control, and ramp metering.
- Rapid response to incidents and collaborative action on the part of various transportation management organizations to provide integrated responses.

- Proactive traffic management strategies including route guidance and pre-trip planning.

Of these issues, the first has already been deployed in TMCs; while the latter three have been implemented but not yet deployed.

1.2.1 Surveillance Systems

Traffic surveillance is an essential TMC system. Traffic information is collected using a variety of technologies: loop detectors, video detection, infrared sensors, vehicle probes, aerial surveillance, etc. The information is captured by the sensors, processed locally, and aggregated for transmission to the TMC. Regular scanning frequencies are 1/240s and broadcasting frequencies are in the order of 1 or 2 seconds.

1.2.2 Control Systems

One of the fundamental properties of a TMC is the capability for controlling a traffic network. Control may be centralized, distributed, or hierarchical. In a centralized environment, a central facility collects traffic status data and makes traffic control decisions. In a distributed environment, control is performed locally, generally at the intersection level. A hierarchical control configuration is a hybrid between central and distributed control. In this architecture, control is generally performed locally; control decisions are monitored by a central facility that may override local control to achieve optimized traffic flow on a sub-region basis.

Regardless of the specific architecture, the vision is for integrated, proactive control rather than reactive control. Proactive control requires the support of a prediction system. A wide range of options is available for the control, including real-time traffic adaptive signal control, adaptive freeway control including ramp metering, transit and emergency vehicle preferential treatment, and lane usage control.

For most control system configurations, the control software will reside within the TMC as one of the support systems. In the case of distributed control, the TMC functions as a supervisory node with control override capabilities. The design and

redesign of TMCs needs to be sufficiently robust to accommodate variations of the architecture.

1.2.3 Incident Management Systems

One of the primary goals of ITS is to reduce congestion. Since a significant portion of traffic congestion is due to traffic accidents and other incidents, a primary functional requirement of a TMC is the detection and management of incidents. Because of the importance of this function, it is generally treated separately, although it involves elements of surveillance, monitoring, control, and decision support.

Certain information processing capabilities are required for the TMC to provide incident management. These include: integrated data management, real-time traffic model execution, image processing for area-wide surveillance and incident detection, and man/machine interfaces providing transparent access to needed information.

1.2.4 Decision Support Systems

Decision Support Systems extend the TMC's traffic control capabilities to support the decisions operators at TMCs have to make. All of the traffic models and simulations used in the TMC reside in the Decision Support System. Online models are used for developing response strategies. They must execute faster than real-time so that their results can affect decision-making processes occurring in real-time.

FHWA (1993) interviewed the managers from the most important TMCs in the U.S. The requirements for Decision Support Systems that the managers suggested included the following:

- Expert systems to aid in incident detection and management.
- Access to traffic simulation models.
- Evaluation models to support route diversion and route guidance.

Thus, existing traffic simulation models are not used online for three reasons: most models are extremely data-input intensive; the data structures in the models do

not correspond to the structures in the TMC databases; some of the network models cannot be executed and analyzed within the time frame available for decision making. Currently there are no methods for the real-time evaluation of route diversion and route guidance strategies.

The overall scarcity of online models results from the fact that existing models have been developed for offline use and are difficult to integrate within an online environment.

1.2.5 Other Systems

ATMS is the core of ITS. As such, it is the integrating agent for both ITS and non-ITS systems. Within the TMC, interfaces are required to all other ITS systems including ATIS, APTS (Advanced Public Transportation Systems), CVO (Commercial Vehicle Operations) and AVCS (Automatic Vehicle Control Systems). Of these, the strongest coupling is between ATMS and ATIS. Centralized route guidance will require high performance processors and efficient algorithms to be resident in the TMC. The communications load will vary depending on whether vehicles communicate directly with the TMC or through a roadside processor.

1.2.6 Computing Environments

The typical TMC configuration uses a mini-computer (e.g., Concurrent, Perkin-Elmer, Modcomp) to operate the control system software, and a networked group of PCs to provide all other TMC functions. The dominance of PCs in the TMC primarily reflects the fact that historically PCs were cheaper than workstations. Additionally, much of the traffic engineering sofware was originally written under Microsoft's DOS. The prevalence of personal computers has created a de facto standard operating system in TMCs: MS-DOS. Many of the managers interviewed by FHWA (1993) expressed an interest in moving towards UNIX environments. Due to the date of this report, it is likely that the operating system conditions have changed. In fact, many TMCs are now updating their systems to UNIX platforms.

As an example of a recently updated TMC computing environment, Anaheim, CA's traffic control center is designed to manage the surface street network. Their systems run on UNIX workstations. The database is managed by a commercial database manager (Oracle^{TM}). The communications network is based on an ATM infrastructure, designed to be compatible with the existing Teleos ISDN PRI Network established by the Caltrans WAN (Wide Area Network) for video-teleconferencing. The ATM Internet-working infrastructure is linked with the Caltrans District 12 TMC and the City of Irvine ITRAC via an OC3 155Mbps SONET fiber optics network, and with the City of Anaheim TMC via ATM T-1 using T-1 facilities provided by PacBell. The system also includes MPEG 1 video transmission system between the UCI ATMS Laboratories and the Caltrans District 12 TMC, allowing for selection and display of freeway video surveillance cameras within District 12. The TMC is distributed using CORBA (Common Object Request Broker Architecture) to provide to external agents the following services: real-time data (VDS, RMS, CMS), CCTV switching, ramp meter control, and raw field device data.

1.3 Research Objective

DTMS are TMC-independent software systems intended to run in all configurations of TMCs. To accomplish this, the TMC must comply with an adequate generic system architecture that allows an easy integration/interface of both internal systems and external systems. Previous efforts have mainly focused on the development of custom ATMS inside the TMCs, or on stand-alone DTMS. However, there have been no efforts to develop an open standard architecture that allows the integration of DTMS within TMCs.

DTMS work independently of one another and no assumption should be made about the possibility of recoding them (or the TMC) for integration. Aicher et al. (1991) proposed four different levels of integration:

- Coexistence, which is in fact no integration. The subsystems operate independently.

- Unidirectional cooperation, which means a data flow only in one direction, e.g. from traffic control to route guidance. The sending system is active and is operating independently. The receiving system is performing an adaptive strategy based on the current control strategy.
- Multidirectional cooperation, which means a multi-way data exchange among the subsystems, e.g. to and from prediction, route guidance and route control. All subsystems have full knowledge of the current status of the other system but an optimization is performed independently.
- Full integration, which means solution of the control problem within one overall strategy. All models are known by all systems. This means that no cooperation rules are needed. The full integration is practically unfeasible because it requires recoding of all the subsystems to know each others logic.

There are logical and physical architectural problems that affect the integration of DTMSs into TMCs. The system architecture defines the physical means of interaction among different platforms (combinations of hardware and software), and deals with issues such as distribution, communications, location services, and concurrent programming. The application design defines the logical relations among subsystems and the work-flow of the system as a whole.

The objectives of a system architecture are to control the complexity and to define the means of distributing the work among the different subsystems. The distribution of the system into subsystems increases modularity and isolates implementation dependencies (Yourdon et al. (1995)).

In this research, a system architecture is developed that allows the integration of various Dynamic Traffic Management Systems under multidirectional cooperation is developed. The architecture is *distributed*, *parallel*, *decentralized* and *open*. Distribution mechanisms will be assured using the Common Object Request Broker Architecture (CORBA). Parallelism is assured through a custom mechanism based on the *push model*[2] that extends the CORBA basic synchronous communication

[2]In a *push model*, servers call the clients, whereas in a *pull model* –the CORBA standard– the clients call the servers (see section 3.4.2 for a detailed development of a custom push mechanism).

paradigm. The architecture allows the design of decentralized systems thanks to the interaction of selected *design patterns*[3]. Finally, the core of the architecture is generic, and permits integration of any DTMS by the development of *adaptors* between the DTMS and the core of the architecture. The main contribution of this research is a core system architecture that allows the integration of different DTMS into Traffic Management Centers. The core system architecture is the result of the integration of design patterns for *location* (Registry), *creation* (Abstract Factory), and *parallelism* (Push).

1.4 Literature review

As seen in section 1.2, Traffic Management Centers are responsible for the management of the operations of one or more regions of the traffic network. These operations include collecting data, activating sensors under a certain strategy, providing guidance to travelers, generating reaction plans to incidents, etc.

There is a lack of literature regarding real-time integration of dynamic traffic control and dynamic route guidance. Current research focuses mainly on the design of independent software solutions (either ATMS or ATIS), and most of the times these have been only tested off-line. There is extensive literature detailing individual models for route guidance and dynamic traffic control.

We will start by reviewing existing traffic control and travel information systems. Then we will analyze previous experiences in terms of combined ATMS/ATIS. We will finish by reviewing some proposed integrated DTMS frameworks. The purpose of this review is to identify the internal subsystems for each of the existing software systems, and to learn from existing frameworks. The literature review focuses on the following elements:

- Dynamic traffic control systems.
- Dynamic traffic assignment for ATIS.

[3] *Design patterns* are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context (see Gamma et al. (1994)).

- Prototypes of combined ATMS/ATIS.
- Design of a generic DTMS, and the study of dynamic route guidance and traffic control, and their consistency.
- TMC architectures.

1.4.1 Dynamic Traffic Control

Gartner et al. (1995) classified the control strategies that can be used to manage a network in five levels. Under each level, a different criterion was used to choose a particular control strategy for the current network conditions. Different levels can be used at the same time for different regions of the network. The levels, in order of increasing complexity, are:

1. Pre-established fixed signal timing plans based on limited access to traffic surveillance and communications.
2. Centralized with on-line optimization (off-line) using a fixed common cycle.
3. Centralized, with on-line optimization, but with variable cycles.
4. Pro-active control based on a prediction (dynamic traffic assignment) module.
5. A combination of all the above. The decision on which levels to use is done through Artificial Intelligence.

The FHWA commissioned the development of a real-time traffic adaptive control system (RT-TRACS) under the Urban Traffic Control System (UTCS) project that started back in the 1970's. Gartner and Stamatiadis (1997) presented the framework for the development of RT-TRACS based on a dynamic traffic assignment module. RT-TRACS is intended to be a multi-level system that invokes a selected control strategy when it can provide the greatest benefits and thus maximize the performance of the system. RT-TRACS will be implemented in an architecture (not defined at this stage) that can interface with other ITS projects. RT-TRACS consists of five basic components. The first component predicts traffic conditions given weather conditions,

type of day, existing flows, incidents, and other traffic factors. The second component defines the sections in the network. Adjacent intersections that need the same control strategy are grouped into sections. The third component selects the appropriate control strategy for the specific section given the traffic conditions that have been predicted. The fourth component implements this strategy, and its performance is evaluated by the final component. There exist several prototypes of RT-TRACS: OPAC (University of Massachusetts at Lowell), RHODES (University of Arizona), and ISAC (Wright State University, Ohio).

1.4.2 Prediction-based (Dynamic Traffic Assignment) ATIS

A TMC can be responsible for providing accurate and up-to-date information to travelers on the state of the network. This information is the result of an Advanced Traveller Information System (ATIS) that uses a guidance strategy. Like control strategies, guidance strategies can also be divided into different levels. The guidance can be reactive (based on current conditions), which corresponds to the first three levels of control, or proactive (based on future conditions), which corresponds to the last two levels of control. The information provided can be descriptive (only recommendations) or prescriptive (regulated). There can be multiple objectives in providing guidance, the most common ones being user equilibrium (each user perceives that he is minimizing his travel time) and system optimal (the sum of all users travel times is minimized).

Dynamic traffic assignment (DTA) has been one of the most recent developments receiving extensive attention in the transportation research communities worldwide (DYNA, 1992-1995; FHWA, 1995; Mahmassani et al., 1994; MIT, 1996). DTA aims at providing route guidance and traffic control based on predicted rather than historically measured traffic conditions.

The Federal Highway Administration (FHWA) sponsors and funds the DTA program initiative. The program is developing a DTA system that can be integrated into TMCs to provide dynamic route guidance and dynamic traffic control.

One of the systems funded under FHWA's DTA program and controlled by ORNL

is DynaMIT (Dynamic Traffic Assignment for the Management of Information to Travellers). DynaMIT (MIT, 1996; Ben-Akiva et al., 1997) is a real-time system designed to reside in TMCs for the support of ATIS operations. DynaMIT is a system that generates dynamic guidance based on a predicted network state. DynaMIT runs in a rolling horizon scheme: the system generates a prediction-based guidance for a prediction horizon starting from the current time. Once an iteration is finished, a new one starts from the current time with a new prediction horizon. At each iteration the system runs two processes sequentially: an estimation (optional) and a prediction-based guidance generation. The estimation process obtains the best available description of the current network state based on the available information. The estimation uses two modules: a demand simulator to model the choices made by travelers based on their habitual choices and the broadcasted guidance; and a traffic supply simulator to move the travelers in a simulated network (using the control strategy currently deployed to modify the capacities of the network). The prediction generates a consistent prediction-based guidance taking the estimated network state as an input. The prediction uses the supply and demand simulator, and a guidance generation module. The result of the prediction is a predicted network state consistent with a dynamic route guidance that can be used to inform the drivers in the network. It is important to note that the control system is an input to DynaMIT. DynaMIT does not model a dynamic control system itself that modifies its strategy based on future network conditions. There are thus two subsystems in DynaMIT that need to be considered in our architecture: a dynamic route guidance system and a prediction system (dynamic traffic assignment based on simulation).

The other system funded under the FHWA DTA program is DYNASMART-X (Mahmassani et al. (1994)). DYNASMART-X provides control actions, in the form of information to users about traffic conditions and routes to follow as well as signal control strategies. The DYNASMART-X system is designed to operate in real-time within a TMC. There are many features that facilitate operation in real-time, including consistency checking and updating modules, to ensure that the model remains in sync with the real world. Origin-Destination estimation and prediction is

the core of the systems ability to predict future traffic conditions. The OD estimation and prediction modules facilitate the ability to generate various control scenarios.

1.4.3 Dynamic Traffic Management Systems

The term Dynamic Traffic Management System (DTMS) is used to designate an ensemble of coordinated subsystems that provide consistent dynamic control and/or dynamic guidance in their operations. These DTMSs are to be installed and integrated within restructured TMCs in the near future. In the most complex case, the TMC will use an intelligent prediction module, e.g. a dynamic traffic assignment module, to generate proactive traffic control and route guidance.

Aicher et al. (1991) studied the possible uses of route guidance data to provide better control management in an integrated system. They proposed a system that would generate a consistent control strategy based on the guidance information.

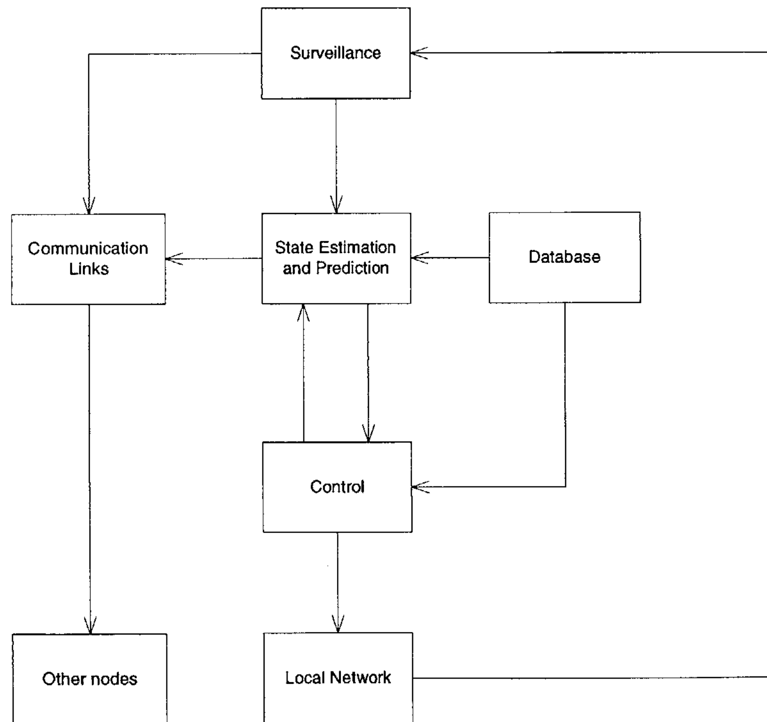


Figure 1-1: Generic Traffic Management Node

Ben-Akiva et al. (1994) presented a nodal real-time DTMS prototype (figure 1-1) for ATMS/ATIS operations based on predicted network conditions. Each node

of the system uses predicted travel time information to generate a control strategy. A simulation based dynamic traffic assignment module is responsible for computing predictions. The node is divided into four subsystems: traffic surveillance, congestion prediction, incident detection, and traffic flow control. The system is hierarchical within different regions of the network, and their nodes were distributed among different processors. The design of each node design is based on a centralized algorithm that controlled the sequential execution of all the subsystems.

Dynamic route guidance and dynamic traffic control need to be consistent with one other to ensure non-contradictory behavior of the integrated system. Chen (1998, 1996) studied the integration and consistency of dynamic control and dynamic traffic assignment. He formulated the problem as an optimization program that aims at finding a consistent solution. Bottom et al. (1998) proposed an iterative approach to study the transient behavior of the interaction between route guidance and dynamic traffic assignment. Chen (1998) also proposed an iterative procedure to obtain consistent dynamic prediction/control/guidance as represented in Figure 1-2.

The European DRIVE II program sponsored the development of the DYNA (DYNA (1992-1995)) traffic system. DYNA uses both traffic control and travel information strategies. Its subsystems include a monitoring module, a traffic prediction module, and a control strategy generator. The European Union's Fourth Framework project DACCORD extends DYNA to include the implementation and demonstration of a DTMS. The Hague Consulting Group (1997) reviewed the design, implementation, validation, and demonstration of DACCORD in different corridors in Europe. DACCORD includes the EUROCOR traffic control project, and the GERDIEN project for systems architecture and traffic prediction. The DACCORD project aims to develop an Open Systems Architecture (OSA) for inter-urban traffic management.

1.4.4 TMC architectures

There is no previous literature about TMC system architectures. The only current efforts are concentrated in defining a high-level somewhat abstract concept of a system

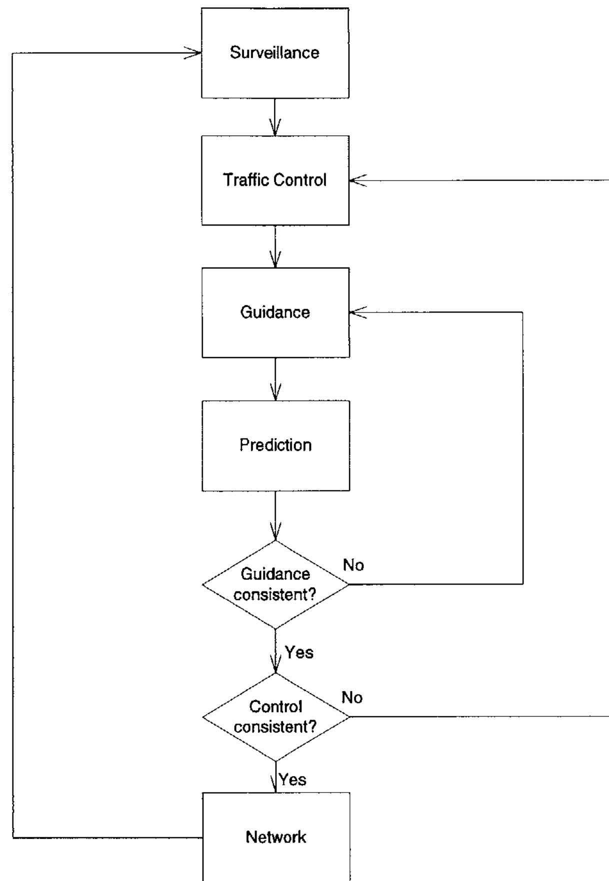


Figure 1-2: Iterative prediction/control/guidance procedure

architecture. Lee Simmons, System Architecture Coordinator for the U.S. DOT's Joint Program Office, and Mike Halladay, ATIS/ATMS Program Coordinator for the JPO are defining what is meant to be a global US National ITS system architecture[4], leading to new standards designed to enhance inter-operability of transportation systems on a national scale.

1.5 Thesis outline

The following chapters will propose and justify a viable architecture that allows the integration of multiple DTMSs into a TMC. The implementation will focus on anticipatory travel information and dynamic route guidance systems, but the design will also include dynamic control systems. To develop this architecture, we used the Unified Software Development Process methodology (Jacobson et al. (1998)) along with the Object Management Group's (OMG) Unified Modeling Language (UML) (OMG (1998)).

Chapter 2 identifies all system requirements, and chapter 3 presents a system architecture that satisfies these requirements, dealing with issues such as object location, concurrent programming, synchronization, deadlocks and distribution. Finally, chapter 5 shows a prototype system that has been implemented based on the proposed architecture. This prototype system integrates DynaMIT, a prediction-based guidance system for ATIS developed at the MIT Intelligent Transportation Systems Program, into MITSIMLab, a simulation laboratory for the evaluation of DTMS, also developed at the ITS Program at MIT. For our purposes, MITSIMLab will play the role of the "real-world" and of a Traffic Management Center. MITSIMLab is composed of two main subsystems: MITSIM and TMS.

MITSIM is microscopic traffic simulator that can simulate the operation of any real world network. It includes a detailed simulation of each individual driver's behavior. It also includes the operations of the surveillance sensors, traffic control devices, and route guidance devices. The control of all these devices is monitored by TMS. TMS

[4]http://www.itsonline.com/archls.html

simulates the operations of a real world Traffic Management Center by collecting data from the sensors, activating signals, and by providing traffic information to the drivers. TMS can interface with external support systems that provide route guidance and/or traffic control. DynaMIT will be integrated into TMS to provide prediction-based route guidance support.

Chapter 2

Requirements

In this chapter, we describe the TMC subsystems and their functionality to derive the requirements for an integrated system.

2.1 TMC subsystems description

The TMC, viewed as a software system, has two layers (see figure 2-1): a *Communication Layer*, and a *Logic Layer*. The Communication Layer is responsible for the activation and operations of the physical devices in the network. These physical devices are: sensor devices (loop detectors, video cameras, probe vehicles, etc.), control devices (signals, ramp meters, lane use signs, etc.), and route guidance devices (variable messages signs, on-board computers, radio, etc.). The Communication Layer is a software abstraction of these physical devices, and assures that the Logic Layer is independent of the actual physical devices used in the network. The Communication Layer is divided into three subsystems that interface between the physical devices and the Logic Layer: *Surveillance Interface*, *Traffic Control Interface*, and *Route Guidance Interface*.

The Logic Layer provides the logic operating these devices. There is one Logic Layer subsystem per Communication Layer subsystem: *Intelligent Surveillance*, *Intelligent Traffic Control*, and *Intelligent Route Guidance*. Additionally, there is an auxiliary subsystem: the *Prediction*.

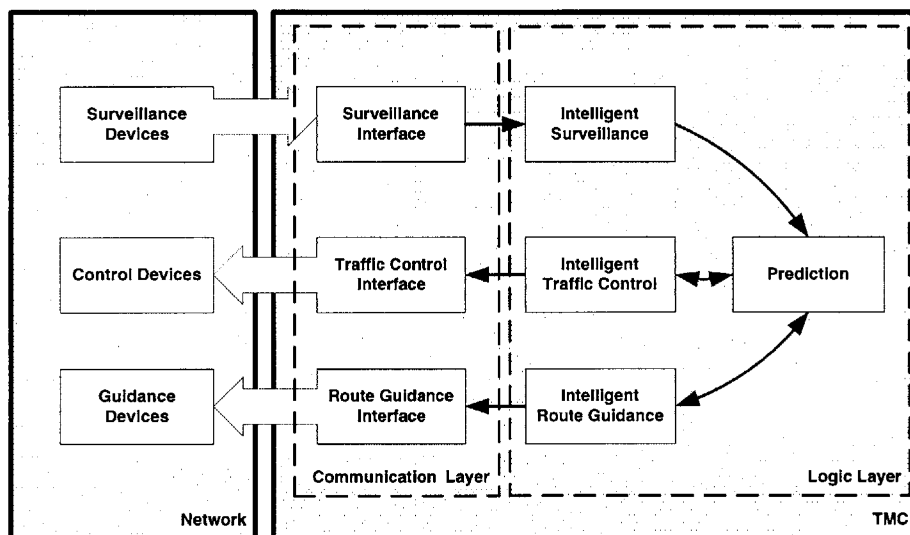


Figure 2-1: Connection and Logic Layers of a TMC

A TMC might have several Communication Layers and several Logic Layers operating hierarchically and in parallel. This is the case, for example, of a single TMC managing two regions of the network under different strategies.

2.1.1 Communication Layers

Surveillance Interface

The Surveillance Interface is a software system that controls the operations of the physical devices and the communication links which collect data from the network. The polled data can be activation times, video images, verbal reports, etc. The characteristics (format, frequency, etc.) of the collected data will depend on the actual network configuration and the type of devices used for surveillance. The surveillance Interface controls the status of the sensor devices, receives the broadcast data from the devices, and transforms it to physical measures that can be used in the algorithms in the Logic Layer. These three tasks will vary from system to system depending on the physical network.

The outputs of the Surveillance Interface are raw sensor readings and possibly incident reports.

A network may be divided into small regions, each of them reporting to a different Surveillance Interface.

Traffic Control Interface

The Traffic Control Interface is, like the Surveillance Interface, the software system that operates the physical control devices in the network. The Traffic Control Interface "translates" the current control strategy into physical activation signals for the devices in the network. The way this "translation" is done depends on the particular control devices used.

The input to the Traffic Control Interface is a control plan generated by the Intelligent Traffic Control system. The outputs are electric signals which in turn activate the control devices.

A network may be divided into small regions, each of them controlled by different Traffic Control Surveillance Interfaces.

Route Guidance Interface

Similar to the Surveillance Interface and the Traffic Control Interface, the Route Guidance Interface is the software system that controls the physical devices in the network used to provide information to the travelers.

The Guidance Interface is responsible for the "translation" of the guidance strategies into the electrical and physical signals needed to activate the guidance devices in the network. The actual implementation of this interface system is dependent on the guidance devices used in the network.

The input to the Route Guidance Interface is a guidance plan generated by the Intelligent Route Guidance system.

A network may be divided into small regions, each of them controlled by a different Route Guidance Interface.

2.1.2 Logic Layers

Intelligent Surveillance

The Intelligent Surveillance system analyzes the data provided by the Surveillance Interface and generates corrected and reliable data that can be used by the other subsystems in the Logic Layer. The analysis consists of searching for erroneous records, missing readings, etc. An important part of the Intelligent Surveillance is incident detection. An incident report can be either an input to the Intelligent Surveillance (for example, a police report that comes through the Surveillance Interface), or it can the result of an intelligent algorithm running within the Intelligent Surveillance system. When the Intelligent Surveillance system discovers the existence of an incident in the network, it immediately makes this information available to the other systems.

The inputs are raw sensor readings and incident reports. The inputs may come

from one or more Surveillance Interfaces. There can be more than one Intelligent Surveillance system.

The output of the Intelligent Surveillance is processed sensor readings and incidents reports.

All of the subsystems in the Logic Layer need the current time in their execution. A separate system must be responsible for keeping the updated time. The Intelligent Surveillance is best fitted for this purpose because it is the only system that has only outputs, and it is the reference that all of the other systems use as their principal input. Hence, the Intelligent Surveillance system will also provide an updated time signature.

Intelligent Traffic Control

The Intelligent Traffic Control system has two responsibilities: to generate a traffic control plan, and to command the Traffic Control Interface to deploy this plan. The traffic control plan can be based on any of the levels in the multi-level design proposed by Gartner et al. (1995). On the first level, the plan is pre-fixed based on historical data and does not use the network state. On the second and third levels, current network states are used to generate a reactive control plan. On the fourth level, a prediction system is used to generate a pro-active control based on future conditions.

The outputs from the Intelligent Traffic Control are a control plan to be deployed by a Traffic Control Interface, and a proposed control plan to be used in the prediction. Note that there can be multiple Traffic Control Interfaces, one for each region of the network.

The inputs are current measured conditions (sensor readings, incident reports and time signatures), and predicted conditions.

Intelligent Route Guidance

The Intelligent Route Guidance system is very similar to the Intelligent Traffic Control. It also has two responsibilities: to generate a route guidance plan, and to command the Route Guidance Interface to deploy the plan. Like the traffic control

plan, the route guidance plan can be prefixed based on historical data and not on the current network state. It can be a reactive plan that uses current network states, or it can be a proactive plan based on the future network state.

The outputs from the Intelligent Route Guidance system are a route guidance plan to be deployed by a Route Guidance Interface (note that there can be multiple Route Guidance Interfaces, one for each region of the network), and a proposed guidance plan to be used in the prediction.

The inputs are current measured conditions (sensor readings, incident reports and time signatures), and predicted conditions.

Prediction

The Prediction system is responsible for calculating the future network state. The Prediction system can be based on statistical inference, dynamic traffic assignment (either simulation based, analytical, or hybrid), or any other prediction method available. In any case, the output of the prediction system is a description of the network state for a period of time (horizon). For example, the horizon can start with the current time and extend to the next 15 or 30 minutes. The state conditions is a time dependent description (for example, every minute) of the network in terms of densities, flows, speeds, queues, and travel times for every link in the network and for the whole horizon. The inputs of the system are the current surveillance data (sensor readings, current time and incident reports), the currently deployed traffic control plan, the currently deployed route guidance plan, a proposed traffic control plan, and a proposed route guidance plan.

2.2 Architectural requirements

The knowledge of the different TMC subsystems and their interactions lets us derive a list of requirementcs of a system architecture. A system architecture defines the means of distributing the work across the different subsystems. It provides the framework wherein various application designs (different instances of the TMC-

integration problem) are supported without any modification to the architecture.

In the case where the Intelligent Traffic Control and Intelligent Route Guidance systems are dynamic (based on predictions), the proposed traffic control and the route guidance plans will change as the prediction changes. Conversely, the prediction will change when the proposed control and guidance plans change. This suggests that the architecture requires a mechanism to reach consistency in terms of prediction/control/guidance.

Chen (1998) and Ben-Akiva et al. (1994) proposed iterative procedures to obtain consistent dynamic prediction/control/guidance (see figures 1-1 and 1-2). These iterative procedures assume the existence of a centralized module to synchronize the sequential execution of the Prediction, Traffic Control, and Route Guidance systems. The only way a module can synchronize the execution of the other systems is to code all the software subsystems under the same philosophy (distribution mechanism, location mechanism, internal logic, etc.). Our objective is to design an architecture that allows the integration of different subsystems, developed under different philosophies, and not to modify the subsystems so that they can be integrated. Hence, the assumption that all of the software subsystems were written under the same philosophy is too strong, and the iterative procedure is practically infeasible, though it is of a great theoretical interest.

It is possible to modify the sequential iterative procedure into a parallel iterative procedure. A parallel distributed design provides the flexibility of integrating different subsystems into a single cooperative one. The best distribution mechanism for our purposes is CORBA (see section 3.2.3). CORBA does not provide a good mechanism to implement concurrent programming, and this justifies the use of a specific asynchronous mechanism as proposed in section 3.4.2.

In the parallel implementation, all of the subsystems (Intelligent Traffic Control, Intelligent Route Guidance, and Prediction) run concurrently, continuously generating new results. Each subsystem has its own consistency criterion. Once the Intelligent Route Guidance system reaches consistency (as dictated by its own criterion), it will command the Route Guidance Interface to deploy the guidance plan.

The Intelligent Traffic Control system works exactly in the same way as the guidance, and once consistency is internally reached, it commands the Traffic Control Interface to deploy the current control plan. The Prediction somehow works differently. Once its own consistency is reached internally (which implies that the control and the guidance have reached their own consistency), the Prediction starts a new horizon based on the current time (the Prediction works in a rolling horizon basis). This design allows one subsystem to reach consistency faster than another one, depending on the complexity of the plan, and the level of the strategy.

Until consistency is reached, the Intelligent Route Guidance (or the Intelligent Traffic Control) generates a new proposed guidance (or control) plan every time there is a new prediction available. Conversely, the Prediction system generates a new prediction every time a new proposed guidance or control plan is available.

Based on these considerations, and the study of the different subsystems done in previous sections, we can identify all the architectural requirements. Table 2.1 summarizes the requirements discussed in this section and their interactions.

REQUIREMENT	IMPORTANCE	RELATED REQUIREMENTS
Open	Very high	Distributed, multi-platform, standard
Concurrent	Very High	-
Standard	High	Open
Anonymous	High	Dynamic location
Dynamic location	High	Anonymous, distributed
Distributed	High	Multi-platform, dynamic location
Multi-platform	Medium	Standard, dynamic location
Object-oriented	Medium	-
Secure	Medium	-
Inter-operable	Low	Standard

Table 2.1: Requirements

A detailed explanation of each of the above requirements follows:

(a) *Open*: The basic requirement for any architecture designed to support multiple systems is to be open. Open architecture implies an easily expandable design that can be adapted to very different applications. As shown in section 1.2, TMCs do not share a standard design form. A system architecture that allows

the integration of generic DTMSs within these different TMC designs needs to be open.

(b) *Anonymous*: Related to the open requirement is the anonymity requirement. Because DTMSs and TMCs can take any generic form, it is important that there is no "hard-coded" reference to any of the systems. The objective is to define an architecture that allows any part of the system to identify its support subsystem as a provider of a certain support function (e.g. surveillance, route guidance, etc.). Furthermore, it is needed that this identification is done at run-time to provide more flexibility. A TMC can have different traffic control softwares in place. Whether which one is being currently used should be transparent to the other subsystems. The systems do not know the name of their support systems, they just know how to find a system that can provide their needed support functions. Of course, anonymity requires a *run-time dynamic location mechanism.*

(c) *Distributed*: Distribution is the third most important requirement in a TMC. Distribution comes in place because the different ITS technologies that support the TMC operations can be located in any platform and any physical location. For example, an ATIS can be located in a traffic center dedicated to travel information; a police or fire department is likely to be remotely located in relation to the TMC. Because the platforms can be varied (as stated in section 1.2), distribution requires *multi-platform support.*

(d) *Concurrent*: Some systems can be considered as dependent systems: their "*father*" system will request a certain service to it and wait for a response. For example, a route guidance system is called every time significant changes occur on the network and new travel information is needed. Some other systems are independent systems: their execution continues even if there is no "*father*" system. For example, the surveillance system continues to operate even if the TMC is not collecting the data. The system architecture must allow different subsystems to run in parallel performing concurrent tasks.

(e) *Object-Oriented*: A desirable feature of a TMC design is to be object oriented

or, at least, to provide object-oriented support. Object-orientation reduces long-term development costs through reusability mechanisms, and reduces development risk through abstraction mechanisms.

(f) *Secure*: Due to the critical nature of the functions performed in a traffic control system, the operations of the TMC and its subsystems need to be secured against information leaks and possible hacks.

(g) *Standard*: It is advisable that the architecture follows the current software standards and leading technologies to insure the widest possible application. An standard system can be more easily changed if the standards are followed. In fact, it is desirable that the system is *inter-operable*, meaning that any couple of systems, independently of what is the particular implementation used, should be able to interact. Hence, this system architecture does not define the actual implementation, but instead defines a generic standard set of functionalities that allows different implementation to inter-operate.

Chapter 3

System Architecture

The objectives of a software system architecture are to control the complexity and to define the means of distributing the work among the different subsystems. In this chapter we select and design a system architecture that fulfills the requirements identified in chapter 2. We first introduce the notion of distributed object computing and review different distribution mechanisms in order to select the one that is better adapted to our system. We then address specific issues such as concurrent programming, location services, and deadlocks. Finally, we propose a layered architecture that integrates the different subsystems and we show several examples of integrated TMC designs.

3.1 Distributed Object Computing

Object-Oriented technology originated in the 1960's in the work of K. Nygaard and O-J. Dahl of the Norwegian Computing Center based around the Simula-67 programming language. The aim was to support the modeling of discrete event simulations of scientific and industrial processes with direct representation of real world objects.

After a quarter of a century, a general interest for object-oriented technology is expressed by the business world. The main reasons of this interest are certainly the fact that objects reflect the real world, are stable, reduce complexity and are reusable.

In the long term, this reduces the code needed to support a business and the time needed to develop it; it increases the consistency in the applications behavior and improves the quality of information systems.

On the other hand, with the advent and widespread use of personal computers and local area networks, it quickly appeared useful to integrate the powers of centralized and decentralized resources while taking into account their respective advantages. Client-Server models have emerged from this necessity. Techniques have been developed to distribute application functionality between clients and servers. (database processing, business application logic and user presentation) However, people have realized that client-server architectures are definitely not easy to implement. It remains extremely hard to design a high performance client-server application, no matter what application development tool is used.

This naturally suggests that a combination of Object-Oriented and Client-Server paradigms can lead to a very powerful concept. This is the idea behind Distributed Object Computing, and the reason why we identified object-oriented and distributed as two requirements of a TMC system architecture.

3.2 Distributed Systems Technology

Various technologies are available to implement distributed systems: sockets, CGI scripts, CORBA, DCOM, RMI, etc. Orfali and Harkey (February 1998, 2nd edition) reviewed these technologies in order to select one that fitted the needs of business domain web-based solutions.

Based on the requirements listed in section 2.2, we specify a set of requirements in order to choose a particular distribution technology:

- *Object orientation (OO).* Is the technology built around an object model that provides for facilitates encapsulation, abstraction, inheritance, and polymorphism?
- *Object services.* Does the technology use its object model to provide additional object services? Additional object services simplify the development task by

providing well implemented solutions to problems that appear often in multiple domains.

- *Security.* Does the technology support a single, consistent security model? Can this model be centrally administered?
- *Inter-operability.* Does the technology establish some standard for inter-operability across implementations over different languages, platforms and operating systems?
- *User interface integration.* Does the technology provide some sort of consistent user interface definition? Are the service usage and intuitiveness of user interface/interaction supported and ensured by the technology?

3.2.1 Sockets

A socket is an end-point of communication within a program. Sockets are used to make computers communicate. Two sockets are necessary to define a communication channel between two computers. The socket implementation is close to the hardware. As a result, they offer excellent performance and flexibility. However, they make the application implementation dependent, they are not object oriented, and they are time expensive to implement.

3.2.2 HTTP/CGI

HTTP (Hypertext Transfer Protocol) / CGI (Common Gateway Interface) is the predominant model for creating 3-tier (see section 3.5.2) client/server solutions for the Internet today. HTTP provides simple Remote Procedure Call (RPC)-like semantics on top of sockets. CGI is one of the most popular UNIX-based technologies that supply interfaces between browsers and servers on the Internet. It is a standard for running external programs from a World Wide Web HTTP server. CGI specifies how to pass dynamic components to the executing program as part of the HTTP request. For example, it allows answers typed into an HTML form on the client computer to be tabulated and stored in a database on the server-side computer. Commonly, the

server-side CGI program will generate some HTML, which will be passed back to the client's browser. CGI is not a programming language. Various "CGI" scripting programs exist that support different languages. Perl is a common choice for writing CGI scripts under UNIX.

Besides not being object-oriented, CGI is very inefficient, insecure, and does not comply with the anonymity requirement.

3.2.3 CORBA

The Common Object Request Broker Architecture (CORBA) is probably the de facto standard for object distribution. The Object Management Group (OMG (1995)) defined CORBA as an object oriented distribution specification that different vendors can implement. CORBA is language independent, and though it is object oriented, does not limit itself to object oriented implementations.

CORBA is now supported by more than 800 members. The main advantage of CORBA is that it is hardware and software independent. The drawbacks of CORBA are that there is not yet a complete definition of a communication architecture and many issues are being resolved independently by different vendors. The applications developed under different vendor CORBA implementations are not portable. New specifications of CORBA tend to define a common set of functionality supported by all vendors.

Using CORBA should be considered when dealing with the following problems:

- *Cross-platform environment.* If the system is developed on a heterogeneous network, CORBA provides an abstraction layer that allows the design and development of a system in multi platform environment.
- *Legacy system encapsulation.* Because of the intrinsic definition of CORBA, legacy systems can be easily integrated into complex open distributed systems, where the clients do not require to know any implementation details about the legacy systems.

As described in 1.2, a TMC can be understood as the integration of different

legacy subsystems. Usually, these subsystems run on different machines and different platforms. CORBA provides a natural way to handle the compatibility issue without incurring expensive reengineering costs: a layer of abstraction between the implementation and the use of a subsystem allows the programmer to easily incorporate changes in the implementation without modifying the interface of the system. Finally, CORBA standards define a Security mechanism that it is natively integrated in the implementations.

The OMG specified the Interface Definition Language (IDL) as standard way to define distributed objects that can then be used from any platform/machine combination. The IDL definitions by themselves are not sufficient to provide remote access. CORBA defines further mechanisms in order to allow remote object access. An Object Request Broker (ORB) is responsible for interfacing the object requests with the physical network layer. Object adaptors are responsible for interfacing the objects requests from the clients to the servers. Finally, additional services simplify the way communications are implemented among the objects, such as the Event Service, the Naming Service, and the Trading Service. These CORBA extensions will be described in section 3.4.

Hence, CORBA complies with all our pre-specified requirements for a distributed technology.

3.2.4 DCOM

DCOM (Distributed Component Object Model) is another industry ORB. It is the distributed extension of the Component Object Model (COM) that developed from Microsoft's work on OLE (Object Linking and Embedding) compound documents. The full set of these technologies is now known as ActiveX. Microsoft calls DCOM "COM with a longer wire" because it uses the same methodology to talk across networks that is used to run interprocess communication on the same machine. Similar to CORBA, this process involves creating a proxy for the object on the local machine and a stub on the remote machine. The proxy communicates with the stub, handling all details. As far as the COM object is concerned, it is working

with an object running in the local address space. DCOM currently is completely available only on two operating systems: Windows 95/98 and Windows NT 4. It is also the foundation technology for ActiveX and Microsoft's Internet. Microsoft's Visual J++ introduced DCOM for Java; it lets a Java object remotely invoke another Java object using the DCOM ORB. Like CORBA, DCOM separates the object interface from its implementation and requires that all interfaces be declared using an Interface Definition Language (IDL). Microsoft's IDL is based on the Distributed Computing Environment (DCE) and not on the OMG's IDL. DCE is an industry-standard, vendor-neutral set of distributed computing technologies. It provides security services to protect and control access to data, name services that make it easy to find distributed resources, and a highly scalable model for organizing widely scattered users, services, and data. DCE runs on all major computing platforms and is designed to support distributed applications in heterogeneous hardware and software environments. DCE is a key technology in three of today's most important areas of computing: security, the World Wide Web, and distributed objects.

DCOM has three main disadvantages when compared to CORBA. Firstly, it is defined and controlled by a single vendor (Microsoft), which greatly reduces the options DCOM developers can choose from (tools and features, for example). Secondly, DCOM's limited platform support restricts the reuse of legacy code and the scalability of DCOM applications. Finally, DCOM is a very immature technology when compared with CORBA. Although Microsoft is currently adding support for DCOM messaging and transactions, these services have been a part of CORBA 2.0 since 1994.

3.2.5 RMI

RMI (Remote Method Invocation) is a native Java ORB from JavaSoft. RMI is a formidable competitor to CORBA because it is part of JDK 1.1 (Java Development Kit) and it is also very simple. On the other hand, RMI does not possess language independent interfaces, therefore it is out of the scope of evaluation as a possible platform for Open architecture components (see section 2.2).

3.2.6 Conclusion

After reviewing the available distribution technologies, CORBA appears to be the best suited for our requirements. Table 3.1 summarizes the different alternatives and the requirements they meet.

	OO	Object Services	Security	Inter-operability	Easy Use	Performance
Sockets						✓
HTTP/CGI				✓	✓	
CORBA	✓	✓	✓	✓	✓	✓
DCOM	✓	✓	✓		✓	
RMI	✓	✓	✓		✓	✓

Table 3.1: ORB comparison

3.3 CORBA overview

The Common Object Request Broker Architecture (CORBA) is structured to allow integration of a wide variety of object systems. The motivation for some of the features may not be apparent at first, but as we discuss the range of implementations, policies, optimizations, and usages we expect to encompass, the value of the flexibility becomes more clear.

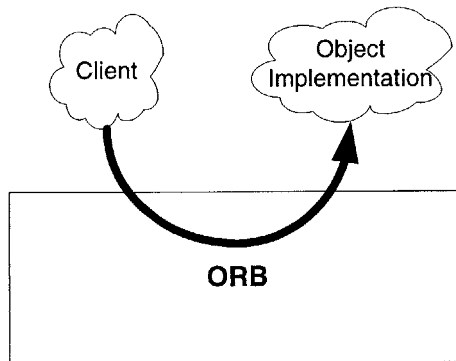


Figure 3-1: A request being sent through the Object Request Broker

Figure 3-1 shows a request being sent by a client to an object implementation.The Client is the entity that wishes to perform an operation on the object and the Object Implementation is the code and data that actually implements the object. The ORB

is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language was used to implemented it, or any other aspect which is not reflected in the object's interface.

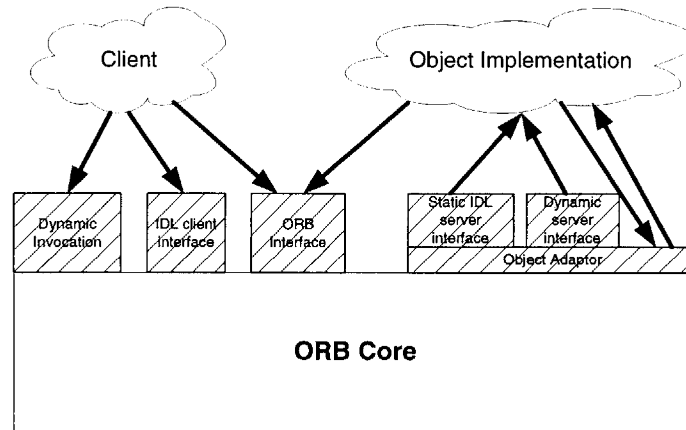


Figure 3-2: The structure of Object Request Broker interfaces

Figure 3-2 shows the structure of an individual Object Request Broker (ORB). The interfaces to the ORB are shown by striped boxes, and the arrows indicate whether the ORB is called or performs an up-call across the interface.

To make a request, the Client can use the Dynamic Invocation interface (the same interface independent of the target object's interface) or an OMG IDL static interface (the specific static interface depending on the interface of the target object). The Client can also directly interact with the ORB for some functions.

The Object Implementation receives a request as an up-call either through the OMG IDL generated static interface or through a dynamic interface. The Object Implementation may call the Object Adapter and the ORB while processing a request or at other times.

Definitions of the interfaces to objects can be defined in two ways. Interfaces can be defined statically in an interface definition language, called the OMG Interface Definition Language (OMG IDL). This language defines the types of objects according

to the operations that may be performed on them and the parameters to those operations. Alternatively, or in addition, interfaces can be added to an Interface Repository service (see Figure 3-5); this service represents the components of an interface as objects, permitting runtime access to these components. In any ORB implementation, the Interface Definition Language (which may be extended beyond its definition in this document) and the Interface Repository have equivalent expressive power.

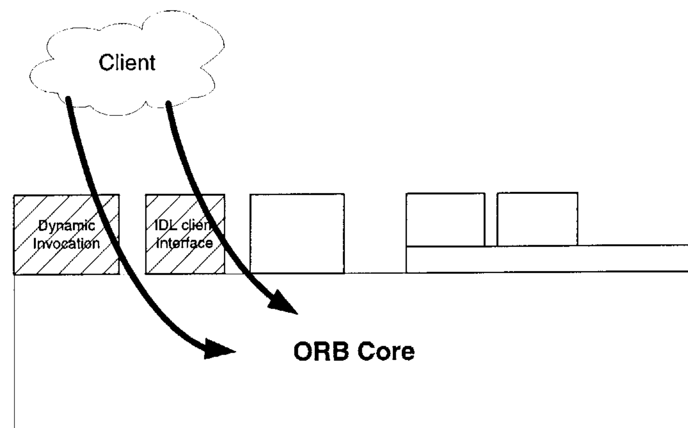


Figure 3-3: A Client using the Stub or Dynamic Invocation Interface

The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. The client initiates the request by calling static interface routines that are specific to the object or by constructing the request dynamically (see Figure 3-3).

The dynamic and static interface for invoking a request satisfy the same request semantics, and the receiver of the message cannot tell how the request was invoked.

The ORB locates the appropriate implementation code, transmits parameters and transfers control to the Object Implementation through an IDL server static interface or a dynamic interface (see Figure 3-4). Static server interfaces are specific to the interface and the object adapter. In performing the request, the object implementation may obtain some services from the ORB through the Object Adapter. When the request is complete, control and output values are returned to the client.

The Object Implementation may choose which Object Adapter to use. This

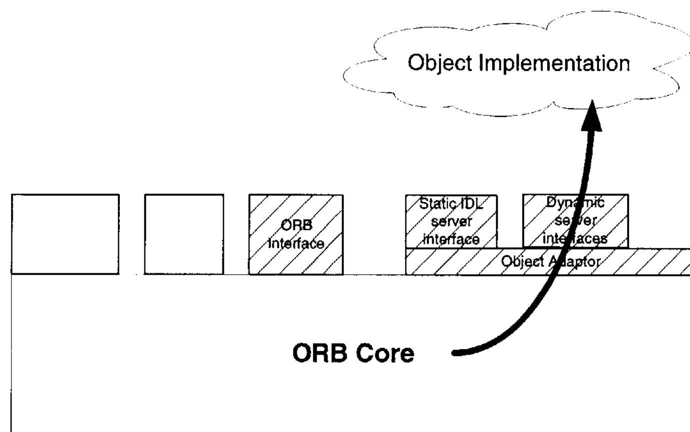


Figure 3-4: An Object Implementation Receiving a Request

decision is based on what kind of services the Object Implementation requires.

Figure 3-5 shows how interface and implementation information is made available to clients and object implementations. The interface is defined in OMG IDL and/or in the Interface Repository; the definition is used to generate the static client interfaces and the object implementation static server interfaces.

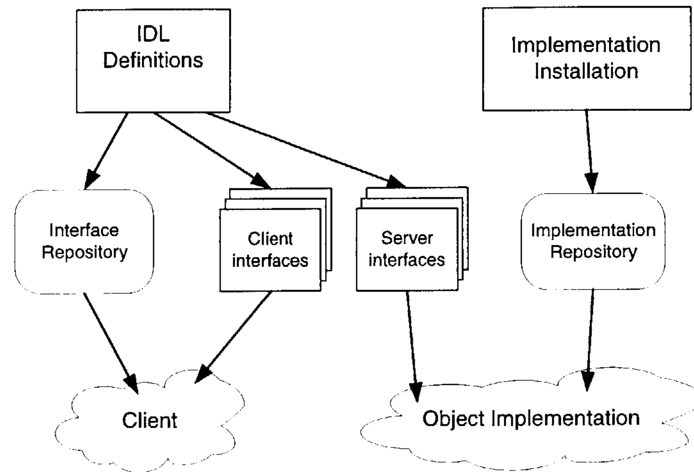


Figure 3-5: Interface and Implementation Repositories

The object implementation information is provided at installation time and is stored in the Implementation Repository for use during request delivery.

3.4 CORBA extensions

Using CORBA as the distribution technology assures that the open, standard, distributed, multi-platform, object-oriented, secure, and inter-operable (only partially) requirements are fulfilled. Some extensions to the CORBA standards are needed to fulfill the remaining requirements for the integration of DTMS within TMCs. The requirements that still need to be addressed are:

- Inter-operability (anonymous, and dynamic location).
- Concurrency.

The main contribution of this thesis is the development of these extensions. They provide a set of tools that when put together with CORBA and object-oriented

paradigms allow us to create a system architecture that fulfills all of requirements established in chapter 2.

The required extensions are explored and defined in the next paragraphs. The combined integration of these extensions (design patterns) into a system architecture will be discussed in the next section. Appendix A shows the complete UML object diagram for the proposed architecture.

3.4.1 Inter-operability

Inter-operability is one of the most desirable requirements for an integrated TMC system. However, it is also one of the most difficult technical aspects encountered in distributed technologies. As such, computer scientists have defined different protocols and applied multiple design patterns in order to be able to provide this functionality. CORBA also provides this functionality, but it needs significant extensions in order to adapt it to a TMC design.

As introduced in section 3.2.3, an Object Request Broker (ORB) is a software package that implements all or part of the CORBA specifications, enabling a client to invoke operations on a remote object regardless of their proximity. CORBA defines separate mappings for different implementation languages (C++, Java, Phyton,...). Furthermore, every ORB software vendor internally implements the CORBA standards in a different way. Because of these differences, there are critical issues in making objects interact with each other under different ORBs. A TMC is the result of the integration of various systems. It is likely that not all of the subsystems were developed to be compatible, and they will probably run under different platforms (hardware/software combinations). Each platform requires a specific ORB, produced many times by different vendors. A system architecture that allows for the interaction of various systems implemented under different platforms is more flexible if it allows more than one ORB.

The two main issues in the interaction of ORBs are:

- *Location mechanism.* In order to use the services of a remote object, a

client first needs to locate the object in the network and obtain a reference to it. This location and binding mechanism was not addressed in the first specifications of CORBA and ORB vendors were then forced to provide their own solutions. Consequently, developers used the ORB-specific mechanisms, making it impossible the port among ORBs. Since CORBA 2.0 specifications (OMG (1995)), standard CORBA services have been defined to facilitate a generic implementation among ORBs. One of these services is the *OMG Naming Service.* However, none of the vendors of the CORBA Naming Service provide access mechanisms to other ORBs as we will see in section 3.4.1. This is the reason we will propose our own location mechanism. This issue is also included in the new CORBA 3.0 specification[1], which is still in review process.

- *Object creation and object use.* The instantiation of an object is achieved differently in every ORB package. The OMG defined what it calls *Object Adaptors*(OA) to unify the approaches taken by the different vendors. Object Adaptors provide a way to create (servers) and use (clients) the objects without knowing the actual implementation technique used by the ORB. Initially, the OMG proposed the Basic Object Adaptor (BOA), but too many important issues were left unheeded, which led to strong incompatibilities among ORBs. The Portable Object Adaptor was the next attempt to provide an OA that was powerful enough to convince ORB vendors to use it. However, at this time, the POA is still immature. We will propose an Abstract Factory implementation that isolates the application code from the creation mechanism. The actual creation mechanism used (POA, BOA, or direct ORB interaction) is not relevant as long as the Factory is the only mechanism used to create an object.

There are more aspects of CORBA that need to be extended besides those that affect the ORB. The OMG is very aware of this and is currently finishing the CORBA 3.0 specifications. In the present standard, CORBA is based on a synchronous communication paradigm. A client that sends a request to a server stays blocked until

[1]Review documents can be downloaded at http://www.omg.org

the server finishes processing the request. However, our software design requires the alleviation of this constraint and allows the client to continue its execution line besides the server's activity. This is called *asynchronous communications* or *concurrent programming* and it is a basic requirement of an integrated TMC system architecture, as identified in section 2.2. CORBA specifications propose several mechanisms to achieve asynchronous communications (e.g. the *Event Service*). We propose here a solution to concurrent programming which is based on the OMG Event Service.

Location mechanism

The clients' first step in using a remote object is to locate the server where this object resides. An easy location mechanism to facilitate such process is needed in any distributed system. The location of the subsystems can be fixed or it can be dynamic. Different TMCs have different subsystems running, and therefore a system architecture should not assume fixed location of the subsystems. The location mechanism must be dynamic and configurable at run time. Furthermore, the location must be implementation independent.

As of CORBA specifications 2.0 (OMG (1995)), most ORBs had their own mechanism to find an object given a name. For example, Orbix (IONA (1995)) uses a static method `CORBA::Object::_bind()`, and MICO (Römer and Puder (1999)) uses a static method `CORBA::ORB::bind()`. These mechanisms are incompatible, and an object created with one ORB can not be located by an object created by a different ORB. In all cases, the vendor specific `bind()` method returns a reference to the object that can be used by the client.

CORBA specifications 2.0 (OMG (1995)) set an optional utility called the OMG Naming Service as a location mechanism that returns a reference to an object given a name. This Naming Service is based on a dedicated server that keeps an active database of the objects' names and locations. Basically, the OMG Naming Service provides a `bind()` operation to bind an object to a name, and a `resolve()` operation that given a name provides a reference to the object. The OMG Naming Service organizes the names in contexts, somehow similar to the way operating systems

organize files in directories.

The normal use of the OMG Naming Service starts with the server instantiating the servant object[2]. Then the server uses the vendor specific `_bind()` operation (it does this either explicitly or internally) to find the Naming Service Root Context. The server will then bind the reference of the servant object to a name in that context. The client first needs to find the Root Context, and then use the `resolve()` operation to retrieve a reference by providing a name. The client can finally invoke operations on this reference.

CORBA specification 2.0 (OMG (1995) introduced a Inter ORB Protocol (IIOP for the Internet, and GIOP for a general protocol) as a way to define the protocol that should be used in the communication among objects running under different ORBs. To be able to use an object running under the responsibility of a different ORB, the client first needs to find the Name Service server where this object is registered. The only way to do this is through the use of the vendor specific `_bind()` operation. Therefore the use of the Name Service as specified by CORBA 2.0 is limited to a single ORB.

As a way to bypass this limitation, many ORBs provide an standard string version of the reference that can be transmitted through a file system or a web server to the client. However, this solution only works with objects that have access to the the same file system.

In the new CORBA specification 3.0[3], OMG is designing the specifications for an Interoperable Naming Service. The CORBA object reference is a cornerstone of the architecture. The Interoperable Name Service defines one URL-format object reference, `iioploc`, that can be typed into a program to reach defined services at a remote location, including the Naming Service. A second URL format, `iiopname`, actually invokes the remote Naming Service using the name that the user appends to the URL, and retrieves the IOR of the named object.

[2]A servant refers to the implementation of a distributed object, whereas the server is the executable program that instantiates the servant.

[3]Review documents can be downloaded from http://www.omg.org, in the process of being reviewed at this time.

For example, an iioploc identifier:

`iioploc://sax.mit.edu/NameService`

would resolve to the CORBA Naming Service running on the machine whose IP address corresponded to the domain name sax.mit.edu (if we had a Name Server running there).

In order to overcome this incompatibility issue before the Interoperable Naming Service becomes available, we propose a location mechanism based on the OMG Interoperable Naming Service. Instead of using an URL-format object reference, `iioploc`, we will use a command line argument to specify the location of our Naming Service.

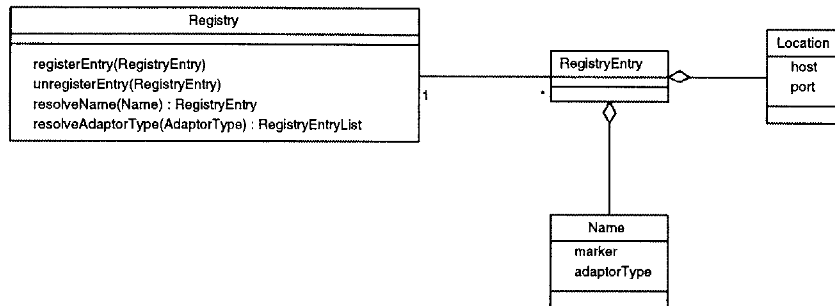


Figure 3-6: The Registry pattern

Our design (see Figure 3-6) is based on an object called *Registry* (similar to the OMG Naming Service server). To activate the service, a server instantiates the Registry object and binds it to a name that the clients can resolve. The Registry contains Entries.

A *RegistryEntry* describes an object and contains a *Name* and a *Location.* The *Name* is a structure that has a string with the entry's name (*marker*) and a variable identifying the role of the object *type*). The *type* is a unique descriptor to identify a group of objects that belong to a same family (e.g. Sources, Listeners, etc.), and can be used to do a search in the Registry for a particular type of object (e.g. search for all route guidance sources and listeners to connect them together). The *Location* is a structure that contains the *host name* where the object is located and the *port* where the server (that is running the servant object) is listening for connections.

The Registry, like the OMG Naming Service server, has a `registerEntry()` operation to insert an entry (a reference to an object) with a particular name and a location, and a `resolve()` operation to retrieve a reference given a name. The Registry also has search functions to find objects of a certain type.

The Registry server executes a periodic consistency check to verify that its entries are still valid, and removes those that correspond to dead objects. All the objects that want to be included in the Registry need to subclass the object RegistryEntry.

Our implementation of the Registry assures that we comply with our initial requirements for a location mechanism. It is dynamic and registers objects at run time, and most of all, it is implementation independent.

Object creation

Object adaptors An object adapter is the primary way by which an object implementation accesses services provided by the ORB. There are a few object adapters expected to be widely available with interfaces that are appropriate for specific kinds of objects. Services provided by the ORB through an Object Adapter often include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations.

The wide range of object granularities, lifetimes, policies, implementation styles, and other properties make it difficult for the ORB Core to provide a single interface that is convenient and efficient for all objects. Thus, through Object Adapters, it is possible for the ORB to target particular groups of object implementations that have similar requirements with interfaces tailored to them.

CORBA standards propose several example object adaptors. The BOA[4] defines an object adapter that can be used for most ORB objects with conventional implementations. For this object adapter, implementations are generally separate programs. It allows there to be a program started per method, a separate program

[4]Note that what IONA Technologies Orbix 2.0 defines as the BOA approach is not really a BOA but an inheritance mechanism.

for each object, or a shared program for all instances of the object type. It provides a small amount of persistent storage for each object, which can be used as a name or identifier for other storage, for access control lists, or other object properties. If the implementation is not active when an invocation is performed, the BOA will start one. The POA (introduced in OMG (1999) - CORBA specification 2.3) originated with the purpose of defining an advanced Object Adaptor that would not change among implementations, so that code can easily be reused for various ORBs vendors. POA is the chosen Object Adaptor for the integrated TMC architecture.

Object implementation CORBA provides two ways of implementing the concrete classes: the first one is by inheritance; and the second one is by delegation, using the *TIE* approach (since CORBA 2.3).

The TIE approach[5] is the most accepted creation mechanism because of the flexibility it gives the developer when transforming a legacy application into a distributed system without changing the class hierarchy.

The latest tendency is to use the TIE approach together with the POA (Schmidt and Vinoski (1998)). The TIE approach is the object implementation mechanism used for the integrated TMC architecture.

Object instantiation The Abstract Factory pattern (Gamma et al. (1994)) provides a way for a system to create objects without specifying their concrete classes. The Abstract Factory isolates the concrete classes from the clients. The clients manipulate only abstract classes. In a CORBA distributed system, the abstract classes are represented by CORBA IDL objects, and the concrete classes are represented by implementation classes. CORBA specification for object creation is based on the Abstract Factory pattern.

In some cases there exists the need to assign the responsibility of instantiating the implementation objects to a dedicated server. A further specialization of the Abstract Factory is provided by a *Concrete Factory*: the Concrete Factory creates objects

[5]The TIE approach was originally introduced by IONA Technologies Orbix and then integrated into the CORBA specification 2.3.

(*Products*) upon requests from the clients. It delegates the actual instantiation into the ORB's Abstract Factory. Clients do not have direct access to the ORB's Abstract Factory. This pattern is shown in Figure 3-7. The Factory has a unique operation: create a product given a name. The Concrete Product in our design is always a RegistryEntry. Whenever a new product is created, it is also added to the registry.

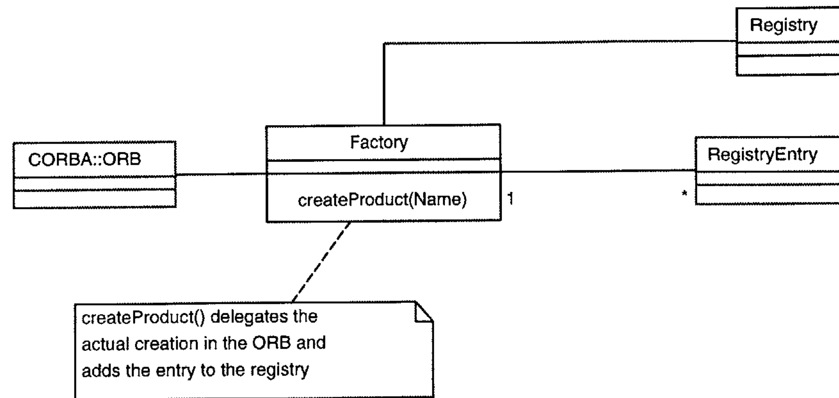


Figure 3-7: The Factory pattern

Our implementation of the Factory relieves the programmers of the subsystems from having to instantiate sources, listeners, etc. Instead, they just need to use the instantiation services provided by the Factory. An example of the use of the Factory in a TMC is shown in section 5.4.

3.4.2 Concurrent programming

Besides the Inter-operability mechanisms studied in the previous sections, concurrent programming is another required function of the system that is not completely provided by CORBA. As identified in chapter 2, concurrent programming is a very important feature of a TMC system. Most of the calls among the TMC subsystems need to be asynchronous. Any system that broadcasts data is performing asynchronous calls to other systems. Also, the core of the TMC requests the DTA system to predict future travel costs in the network. However, the TMC cannot stay blocked while the DTA system completes its execution.

The basic paradigm of CORBA communications relies on a synchronous model.

Clients invoke operations on their proxies, which in turn do the work: construct a Request object, marshal arguments, send over the network, wait for a response, and return it to the client. Meanwhile, the client application code is blocked until completion of the request. A synchronous model is implemented in a centralized manner: all operations are coordinated from one single client.

This requirement is based on the following needs:

- *Zero Waiting Time.* If the server is busy, the client will be blocked. The client might need to continue its execution without waiting for the server to complete the call. For example, the traffic control system cannot stay blocked waiting for a prediction to be available.
- *Multicast.* Some systems in a TMC change their state at an unknown rate (e.g. a prediction system usually takes an indefinite amount of time to generate a new state prediction). Instead of blocking all of the clients until a state change happens or having the clients continuously poll the server, we might want several clients to be automatically informed by the server of the state changes. For example, most subsystems in the TMC need to know current real time. This time is generated by an object in the TMC. Instead of continuously polling the time object, the time can be updated to the clients through a callback.
- *Concurrency.* Several servers might be running in parallel, and a centralized client that controls the execution might not exist, or, even worse, might not be possible to implement. Because of the differences in the logic and design of the subsystems in the TMC, it might not be possible or feasible to design an iterative process that generates a consistent solution (assignment/control/guidance). Parallel cooperative execution might a better solution.
- *Anonymity.* The clients might not know (or the design might not want the clients to know) their servers directly. The clients should not ask the servers directly for services.

CORBA proposes three standard mechanisms that extend beyond the basic synchronous communication paradigm: oneway, DII deferred invocation, and the

OMG Event service. We propose a non standardized mechanism based on the OMG Event service: Pushing.

Oneway

Oneway is merely an IDL keyword that identifies an operation as flowing exclusively in one direction. CORBA specifies that the operation's return value must be void, and that all arguments must be of type "in"[6]. Because CORBA specifications are limited to external interfaces, there is no CORBA requirement for the underlying operation to be non-blocking. CORBA says only that the delivery semantics are "best effort", which implies that the efficiencies of non-blocking sends are allowed, but are not required, leaving the determination up to individual implementations.

If an operation is identified as being oneway in an IDL interface, the IDL compiler can do the checking to ensure that it is only used semantically as a one-directional message.

An ORB implementation can support the language feature while still generating a blocking socket call in the client stub: it is possible to have a congested server with full TCP buffers, so the client will unwittingly send and wait until the server is free. This means the client can be blocked for arbitrary lengths of time (as long as it takes for the server to abate).

On the other extreme, if oneways are implemented as "send and forget" UDP datagrams, reliability is lost: if the server buffer is full, the client will not be blocked, but the message also will not be delivered.

One would like the client to request an operation from the server; do something while the server processes; and then have the server call back with the response. This can be implemented as a double oneway: one "send" operation on the server, and one "receive" operation on the client.

[6] *in* is an IDL keyword that specifies an argument to be only input, but not output

DII deferred invocation

The DII (Dynamic Invocation Interface) acts as a sort of "generic client proxy". A client can use it to bind to any server object without having to have a proxy for that object at link time. The client makes generic operations on the DII interface by packing the arguments needed.

The DII has operations `send_deferred()` and `get_response()` as "built in" operations, which correspond to the send and receive operations in the earlier "double oneway" mechanism.

The problem here is that the mechanism is buried in the DII. That means that all of the work of marshalling the arguments into the `send_deferred()` call, and unmarshalling the results from `get_response()` is left up to the application developer.

Worse, the (efficient) static type checking that static CORBA stubs provide has been lost: how do we ensure that when arguments lists are composed by the client that they correspond to the server's signature? Type problems will appear at run time instead of compile time.

CORBA alleviates this problem by browsing the IR (Interface Repository) to retrieve the correctly typed server operations. But browsing the IR, and relying on it to do strong type checking at run time, is prohibitively expensive, and may still result in runtime type violations, if the client-side developer writes syntactically correct but semantically flawed IR browsing and marshalling code.

OMG Event Service

The OMG Event model is based on the "publish/subscribe" paradigm, a model of distribution which supports highly decoupled interactions between clients and servers, including support for the anonymity requirement. Inserting an Event Channel as a third party between the client and server provides this high degree of decoupling. Note that with a typed Event Channel, the disadvantages noted earlier about the DII Deferred Synchronous connection are removed: compile-time strong type checking is provided.

There are, of course, new performance penalties introduced, though: messages are forced through an intermediary (the Channel) which may perform buffering, message decoding, etc. These penalties do increase the flexibility of the communication. Buffering, for example, can allow large message volumes to accumulate awaiting server readiness, thus removing the need for client-side flow control.

The primary criticism levelled against the OMG Event Service, as currently devised, is that it still only supports a oneway message flow.

Pushing

Pushing (*push model*) is a way to extend the basic CORBA communication paradigm to provide asynchronous communications. It is based on the OMG Event Service pattern, but it does not incur a performance overhead: it does not provide the buffering and encoding functionality the OMG Event Service does. A very simple but clear example of an implementation of a *push model* can be found in Schettino and O'Hara (1998).

In the basic synchronous CORBA communication paradigm, the client *pulls* data from the server, and it remains blocked until the server is done. In a *push model* the server broadcasts information to several clients. To be able to push, the server needs to know its clients: the publisher/subscriber pattern is the basic underlying design under the push model.

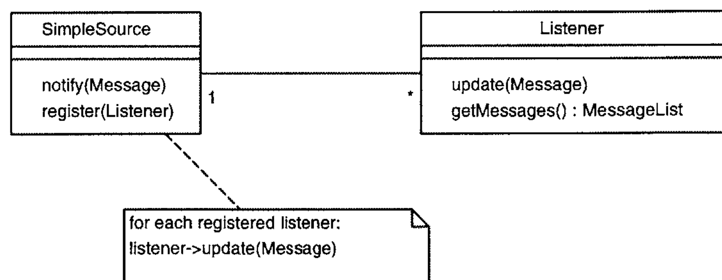


Figure 3-8: The Source/Listener pattern

The pattern is specified in Figure 3-8 (note that instead of Subscriber and Publisher we use the terms Listener and Source since in distributed computing these

terms are more common). The clients (Listeners) register themselves in the servers (Sources). The Sources maintain a list of their Listeners. Whenever there is a change of information that needs to be broadcast, the Source will go through its list of Listeners and invoke their `update()` operation. The broadcast data will be parsed as an argument to the `update()` operation. This broadcast assures the multicast requirement that we set for an asynchronous solution in CORBA.

The Listener and Source need to use threads to be able to assure the zero-waiting time requirement. In this manner the Listener is not blocked by performing another operation when the Source notifies new data, and viceversa: when the Listener processes the message (a remote object for our purposes), the Source is not busy[7] and can provide access to the Message object. Threads are implemented internally in some languages (such as Java), or can be explicitly implemented (such as C++). Note that the use of oneway (see 3.4.2) operations is justified here for the update() operation, because threading assures that the server will not be blocked.

However, the anonymity requirement is lost in this pattern. A way to overcome this problem is to introduce an intermediate server that preserves the identity of the publisher. The introduction of a typed channel between the source and the listeners (the same way the OMG Event Services uses a typed event channel) provides a high degree of decoupling. As in the OMG Event Service, there is a performance overhead associated with the use of the channel. Unlike the OMG Event Channel, our channels are specifically designed for our application. The Channel we are proposing does not have buffering: it does not stock Messages when it does not have Listeners, though it could be extended to do so. Note that in this case, the use of the OMG Event Service would be justified, because it is exactly this additional functionality what distinguishes our Channel from the COS Event Channel.

In Figure 3-9, the object model shows how a Channel can be used by several Sources, and also, how multiple Listeners can register themselves as subscribers of

[7]The Source may be busy if it is answering another request. This is because CORBA 2.3 only allows an object-by-reference as the argument to an operation, and therefore the Listeners use a reference to the message located in the Source and not a copy of the message itself. CORBA 3.0 will add object-by-value and the Source will not need to be threaded.

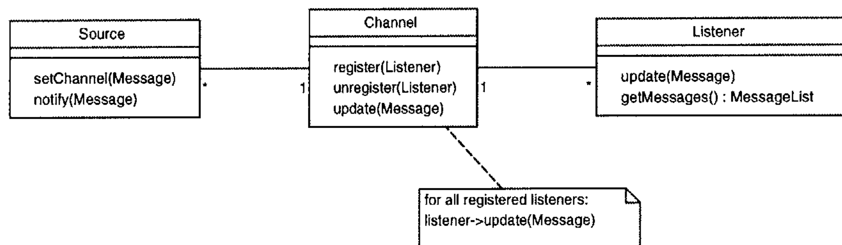


Figure 3-9: The Source/Channel/Listener pattern

this information. The anonymity requirement is assured in all cases.

As an example of how an asynchronous communication occurs with this pattern, suppose that we have a TimeGenerator that broadcasts very frequently an updated time signature . The TimeGenerator server needs to create a Source object. By some external mean the Source will be connected to a Channel using the `setChannel()` operation (e.g. the user can do this through a graphical user interface). On the other side, a TimeConsumer server that wants to be kept updated of the current time will create a Listener and register it in the Channel (again, the registration can be done through a GUI). Whenever the TimeGenerator server has a new time signature to broadcast, it will first construct a Message object, and then call the `notify()` operation in the Source passing a reference to the newly created Message. The Source will call the `update()` operation of the Channel, again passing a reference to the Message object. Finally, the reference to this object will be passed through the `update()` operation of the Channel to all the registered Listeners. Each Listener keeps (buffers) a list of references to the incoming Message objects created in the TimeGenerator server. Whenever the TimeConsumer needs some updated time information, it invokes the `getMessages()` operation of its Listener. This will provide the TimeConsumer with a list of references to the Message objects that reside in the TimeGenerator server. It will also clear the Listener's buffer. Once the TimeConsumer finishes using the references to the Message objects, the objects are destroyed and the memory is deallocated in the TimeGenerator server. The TimeConsumer will probably only use the most updated time signature in the list and discard all the other previous messages. However, the design permits the buffering ofthe messages, something that it is sometimes necessary (e.g. the sensor readings from the surveillance system). Unlike the OMG Event Channel, where the Channels buffer the messages even when there are no objects using those messages, we buffer in the Listeners because they are necessarily linked to a server that uses those messages. Buffering in the Channel might prove useful in some other applications (e.g. a stock trader needs trading information since the beginning of the day, even if he joined after the bell).

A Message can be a very simple object, like a time signature, and include only data members like a sensor reading, or it can contain very complex objects with multiple operations that provide some desired logic, like an algorithm.

Deadlock avoidance

A deadlock defines a situation wherein two or more processes are unable to proceed because each process is waiting for one of the others to complete. Deadlock avoidance is a very important issue in network packet routing and in distributed computing.

In the synchronous communication paradigm, there exists the potential risk of deadlocks. In the simplest case, object A requests services from B and B from A. In the asynchronous case, object A(B) continues its execution after effectuating the request to B(A). Therefore A(B) listens to requests from B(A), and there are no communication deadlocks.

For the synchronous case, we can model the object relations with a graph: consider each object as a node and each request as an arc of a directed graph. To assure that we have no deadlocks, the graph must contain no directed cycle.

For a directed connected graph with n nodes to contain no cycle it must have at most n-1 arcs. A directed tree contains n-1 arcs, hence it has no cycles. Hence, our interest is to develop an architecture that can be represented as a tree (or as a forest, for this purposes).

Peter and Puntigam (1997) proposed an alternative mechanism to assure the existence of no deadlocks. It is possible to assure the existence of no deadlocks by performing static type checking of the objects involved in every asynchronous transaction.

3.5 System distribution

The previous sections have dealt with defining the technical mechanisms to meet all the requirements defined in chapter 2. A software system architecture is intended to use these mechanisms to build a structure on which the application design can be

built. Jacobson et al. (1992, 1998) introduced the construction analogy to describe the mission of a system architecture. One could see the technical mechanisms explained in previous sections as the bricks and beams needed to build a house. But now these bricks need to be put together into the structure of the house. And finally, we can build the rooms.

The first problem when designing a distributed system architecture is defining what parts of the systems are located where and how much they are allowed to interact with the other systems. Basically, the challenge are:

(a) *Distribution pattern*: when designing an object system for a client-server environment what is the most appropriate way to structure the overall application architecture?

(b) *Load balancing*: how does one distribute responsibility among the different subsystems and machines to best take advantage of each platform's unique capabilities?

The four-layer pattern is the most accepted distribution pattern solution. The TMC distribution architecture is a modified version of the four-layer pattern. Tanenbaum (1988) introduced the notion of layered architectures with his seven layers for Operating System Interfaces, now widely used in computer networking, and suggested the use of a four layer architecture for distributed client/servers.

The three-tier pattern is the most widely accepted solution to define load balancing among platforms and subsystems (specially in web based applications). It is a useful pattern that we will use as a reference in the TMC distribution architecture, and that was introduced as an extension of the Model-View-Controler (MVC) pattern by Buschmann et al. (1996).

3.5.1 Four-layer client/server distribution

When designing the software architecture in a client-server system, one must come up with a way to divide the functional responsibilities across the systems and the developers. The architecture must also be simple enough to be easily explainable, so

programmers can understand where their work fits. Based on the layered architecture concept introduced by Tanenbaum (1988), Brown and Whitenack (1996) defined a four layer architecture to distribute the coding work of a large program written in Smalltalk.

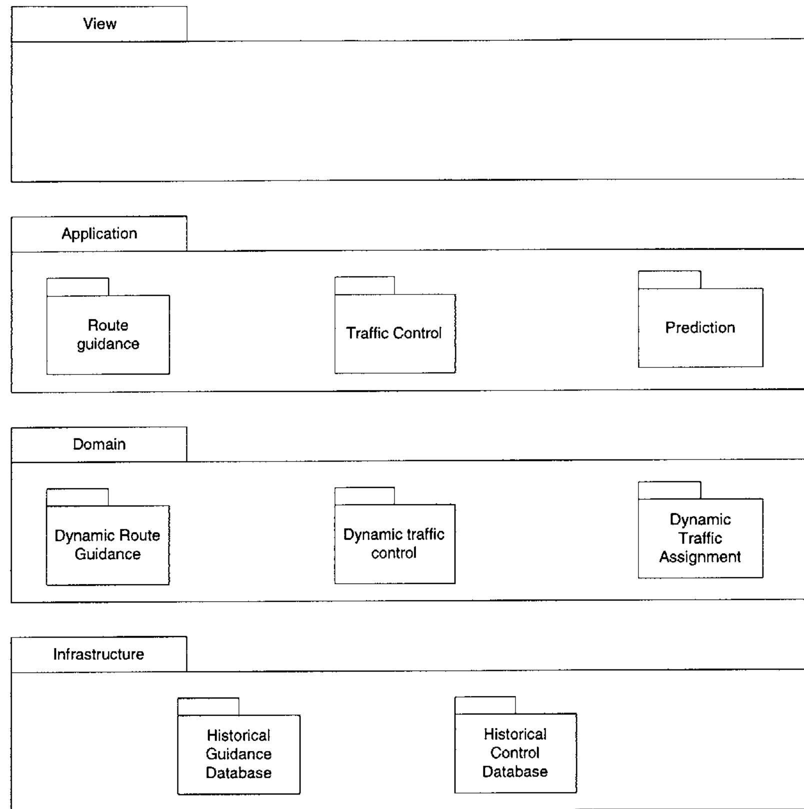


Figure 3-10: Four-layer distribution architecture

A layered system is divided into horizontal and vertical layers. Represented in Figure 3-10, vertical layers are used to isolate functionality, while horizontal layers are used to isolate the system from implementation dependencies.

Vertical layers should be lowly coupled so that there are no dependencies between implementations. The vertical layers are normally identified by functional requirements. Different horizontal layers carry different levels of responsibility within the same vertical layer. The four horizontal levels are:

- The View layer. This is the layer where the physical window and widgets reside. It may also contain Controller classes as in classical MVC. Any new

user interface widgets developed for this application are put in this layer. In most cases today this layer is completely generated by a window-builder tool.

- The Application Model layer. This layer mediates between the various user interface components on a GUI screen and translates the messages that they understand into messages understood by the objects in the domain model. It is responsible for the flow of the application and controls navigation from window to window. This layer is often partially generated by a window-builder and partially coded by the developer. The application layer can be further layered into the view controllers that control the view of a particular client, the use case controllers that direct the flow of events in each of the use cases of the system, and the application-domain adaptors, that translate the data request to a format that the objects in the domain layer can understand.
- The Domain Model layer. This is the layer where most objects found in an OO analysis and design will reside. To a great extent, the objects in this layer can be application-independent. Examples of the types of objects found in this layer may be Network Topology, Surveillance System, Prediction Algorithms, etc. These objects are grouped in packages that pursue a common functionality within the system and that are integrated into a vertical layer.
- The Infrastructure layer. This is where the objects that represent connections to entities outside the application (specifically those outside the object world) reside.

If the design is strict about clearly defining the interfaces between the layers and where the objects fit within those layers, then the potential for reuse of many objects in the system can be increased. It is custom to locate the View and Application layers in the client and the Domain and Infrastructure layers in the servers.

3.5.2 Three-tier distribution

A very simple example of distribution architecture is based on a direct client/server communication where it is possible to locate most of the application code on the

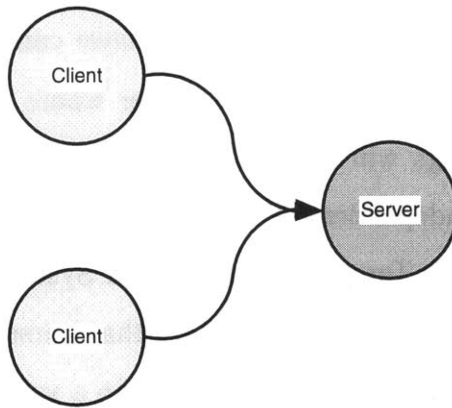


Figure 3-11: Client/server distribution architecture

client. Represented in Figure 3-11, the client has access to a database server that is shared with all other clients using a vendor-supplied API. Even if this architecture is simple to build, it leads to the following problems:

- When locating most or all application code on the clients, they (the clients) have to request or download all the data they need to fulfill their business tasks. This is sometimes too inefficient. The network becomes overloaded when many clients request a great amount of data (or objects). Also if a client does not have the chance to delegate the processing of the requested data to another client, application performance or response time depends heavily on the power/efficiency of the system supporting this client.
- If the system relies on a vendor specific server API, the system will become too inflexible, because it is restricted to a specific set of services the server places to the clients' disposal. The system should try to prevent getting "locked into" a specific vendor.
- Distributing the application code between clients and servers makes it more flexible and scalable, but the larger the number of parts, the more complex a system seems, and the more failure points exist. Therefore, the system becomes also more difficult to maintain.
- If the clients interact using a (non-active) database server, automatic change propagation is unsupported, or at best supported by highly vendor specific database triggers mechanisms. This may require continuous checking (polling) of the server to detect changes, resulting in extra network load. Depending of the polling time-frame some clients become inconsistent with the database for a while or may be affected by network traffic problems.

Buschmann et al. (1996) refined this simple architecture using the MVC architecture in order to develop the well accepted 3 tier distributed architecture. Represented in Figure 3-12, the basic client/server architecture is completed by balancing the load between clients and servers. An additional tier between the database and domain servers is introduced. The new architecture is a 3 tier distributed architecture with modified responsibilities:

- The front-end clients tier. These clients should not code any of the database access. The implementation allows the clients to be independent of the server API. The clients become thin-clients because they do not need to implement any vendor-specific logic. This responsibility is transferred to the application servers.

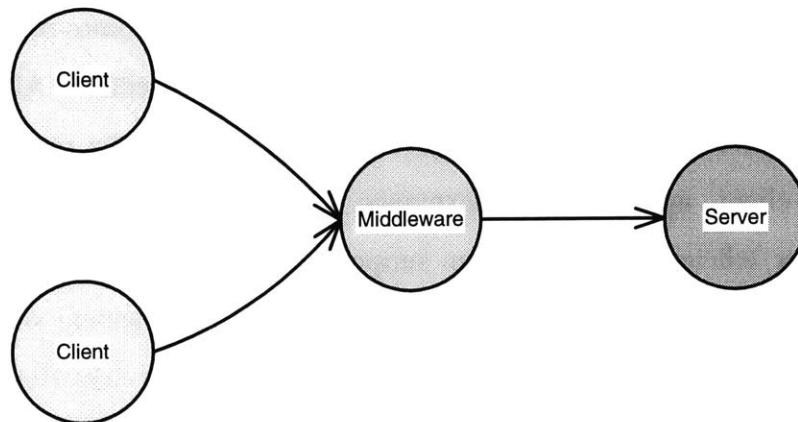


Figure 3-12: Three tier distribution architecture

- The application servers (*middleware*). These servers interface the clients requests to the particular API of the back-end servers. The clients can still be used with different vendor specific database servers. The application servers encapsulate the logic needed to interact within themselves and with the back-end servers. CORBA is an example of middleware.
- The back-end servers and database servers. These servers are now only interacting with the application servers. A substitution on a back-end server, or a modification of the logic only affect the application servers, but not the clients.

This architecture allows the application logic to reside exclusively on the application servers. The back-end servers as well as the front-end clients do not have any knowledge of each other. This assures reusablity, low coupling and low development risk.

The 3 tier architecture is the most accepted distributed framework for web based applications. In a web-based application, the client requests services from a web

server that transmits that request through a middleware to a domain server and/or database server. Once the data is ready or the request is finished the server transmits that information back to the client.

In a TMC the communication is more complex than in a web based application because the servers are also clients, and there is no *pure client.* A TMC consists of different subsystems running in parallel producing messages that are consumed by other subsystems. In this context, the 3 tier architecture assures that none of the subsystems are dependent on the implementation of the other subsystems.

3.5.3 TMC system architecture

A TMC system is the result of the integration of different subsystems. To design such system, we need the support of a system architecture. The system architecture provides us with the means to integrate the different subsystems providing all of the functionality is identified in the requirements.

To create such a system architecture, we use the results developed in previous sections, including the CORBA specifications and the CORBA extensions developed in section 3.4. The result of combining these CORBA extensions (design patterns) with CORBA specifications and a four-layer distribution architecture is the TMC system architecture.

To adapt the four-layer architecture proposed in section 3.5.1 to the TMC design, we must identify what to locate in each layer.

It is possible to understand an integrated TMC as the concurrent cooperative execution of several (legacy) subsystems that produce and consume information from each other. A subsystem can consume and produce different types of information at the same time. Based on this conception, we locate the main functionality of the TMC in the domain layer. The domain layer contains the parts of the system that are permanent to the operations of the TMC, and independent of our architecture. These parts are what we have identified so far as subsystems: surveillance interface, route guidance, traffic control, etc. In their execution, these subsystems need access to complex databases that provide persistence to the TMC. These databases contain

historical data, network topology descriptions, geographical data, sociological data, etc. The databases communicate with the systems in the domain through CORBA, but because these are not execution processes but instead are stores of data, the communications with the subsystems can be synchronous. Hence, in the interaction between domain and persistence layers, we do not need a concurrent mechanisms like the push-model we developed in previous sections.

The application layer contains the set of sources and listeners that connect the subsystems. Subsystems in the domain later are not allowed to interact with each other but through the use of the push-model deployed in the application layer. The application layer contains the set of sources, channels, and listeners needed to allow the communication among systems in the domain layer. Note that this architecture allows the dialogue between the domain and the application layer, something that it is not common in four-layered architectures (where we can only talk from the application to the domain layer). We are allowed to do this here, because we assign the application and domain functionality to the required layers: all the domain objects (subsystems) are located in the domain layer; and our application layer is merely a tool to allow the parallel execution of the subsystems. Hence, we comply with the functional distribution of a four-layer architecture.

In the view layer, there must be a client that controls the execution of the application objects, which in turn control their domain objects. The functional of this client is to coordinate the channels, sources and listeners.

We develop a cooperative four-layer architecture using the pushing mechanism explained in 3.4.2: in the architecture represented in Figure 3-13 each subsystem needs to open as many Source objects as types of messages it produces. It will also need to open as many Listener objects as types of messages it wants to receive. The application layer provides the Channel objects for these messages to go through and the adaptors for the domain layer (Sources and Listeners). The domain layer is filled with the (legacy) subsystems. The infrastructure layer provides persistence and other support to the domain layer. In the view layer there is a user interface to control how the flow of data goes through the channels: the user can control what messages are

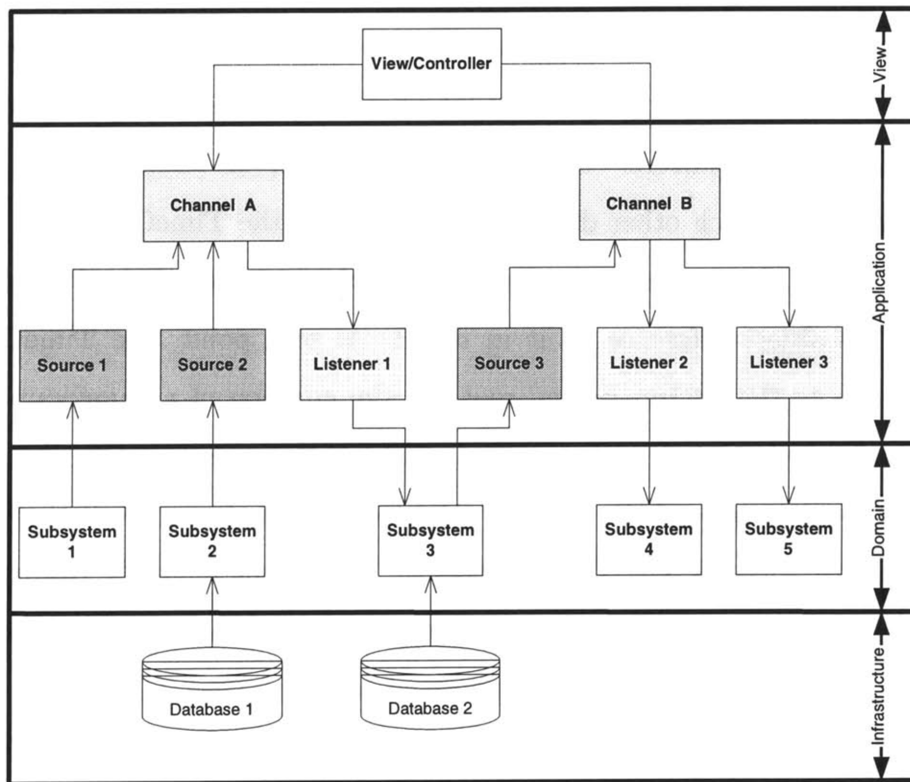


Figure 3-13: Integrated TMC distribution architecture

dispatched to what listeners by controlling the connections among Sources, Channels and Listeners.

Finally, the architecture has a vertical layer that contain a series of tools that permit the execution of the systems. These are the ORB itself, the Registry, and all the necessary Factories.

As an example of execution, we will continue with our TimeGenerator server example started in section 3.4.2. Suppose there is a Registry server running in hostA, and two Factories: FactoryA on hostB and FactoryB on hostC. These two servers constitute the vertical layer. We want to run a TimeGenerator server that continuously broadcasts a time signature, and a TrafficControl server that would like to receive this time signature. Using this architecture, we assure that the servers do not interact with each other directly. After launch, the TimeGenerator finds the Registry in hostA. It then asks the Registry for the location of the FactoryA, and it retrieves an object reference to it in hostB. At this point, the TimeGenerator has a reference to the FactoryA, and requests the creation of a TimeSource server. Automatically upon creation of this server, it is entered in the Registry. Similarly, the TrafficControl server locates the Registry, and from it the FactoryB. It then requests the FactoryB to create a TimeListener server. The client is finally responsible for configuring the servers in the application layer. The client first locates the Registry and does a search for all the servers associated with the type "TIME". It retrieves the TimeSource and the TimeListener. The user can then connect both objects. For this, the client application requests FactoryA (or FactoryB) to create a TimeChannel. The only thing remaining to configure the system is to tell the TimeSource to use the TimeChannel, and the TimeChannel to use the TimeListener.

Every time the TimeGenerator creates a message, it notifies the TimeSource. The TimeSources calls the `update()` method of the TimeChannel, which in turn calls the `update()` method of the TimeListener. The TimeListener finally transmits the time signature to the TrafficControl. The TrafficControl server could save this information in a database. However, the time signature is just part of our example, and the server will normally not save it. Other messages that could be saved in other examples are

traffic counts or travel information.

Chapter 4

Application design

So far, we have identified the requirements of an integrated TMC system, and developed a system architecture that allows us to develop it. We will now use the system architecture developed in chapter 3 to design a TMC system. We will also develop and apply the proposed architecture for other applications designs, such as parallel traffic microsimulation.

4.1 Integrated TMC system design

Each subsystem outputs data consumed by other subsystems, and uses data produced by other subsystems as inputs. The actual inputs and outputs of each subsystem have been defined in section 2.1. In a parallel design, the easiest way to isolate subsystems' logic from each other is to use the pushing mechanism described in 3.4.2. Using that mechanism, each subsystem will create as many Sources as types of data it produces, and as many Listeners as types of data it consumes. Whenever a subsystem internally generates new data, it broadcasts through the Source to all Listeners. Reciprocally, a subsystem gets notified through its Listeners of any change of data from another subsystem.

The Sources and the Listeners are joined together through Channels. Channels, Sources, and Listeners are the core of the system architecture. The TMC design is based on the TMC distribution architecture proposed in section 3.5.3.

The distribution architecture allows any possible set of connections, and accepts centralized, decentralized and hierarchical system designs as shown next.

The next section presents different TMC system designs that can be implemented using this architecture. The graphical representation of a whole TMC design using sources, channels and listeners is too complex. We will graphically represent a simplified version of the design. The simplification consists of substituting the sources, channels, and listeners, by arrows and ovals indicating the message type. For example, Figure 4-1 illustrates the simplification process. On the left of the figure, a complete design model for the interaction of two systems is represented: an Intelligent Surveillance broadcasts sensor readings that are received by a Prediction system. A GUI controls the connections of the sources, listeners and channels. The Prediction system stores the surveillance data in a database for off-line calibration purpose.

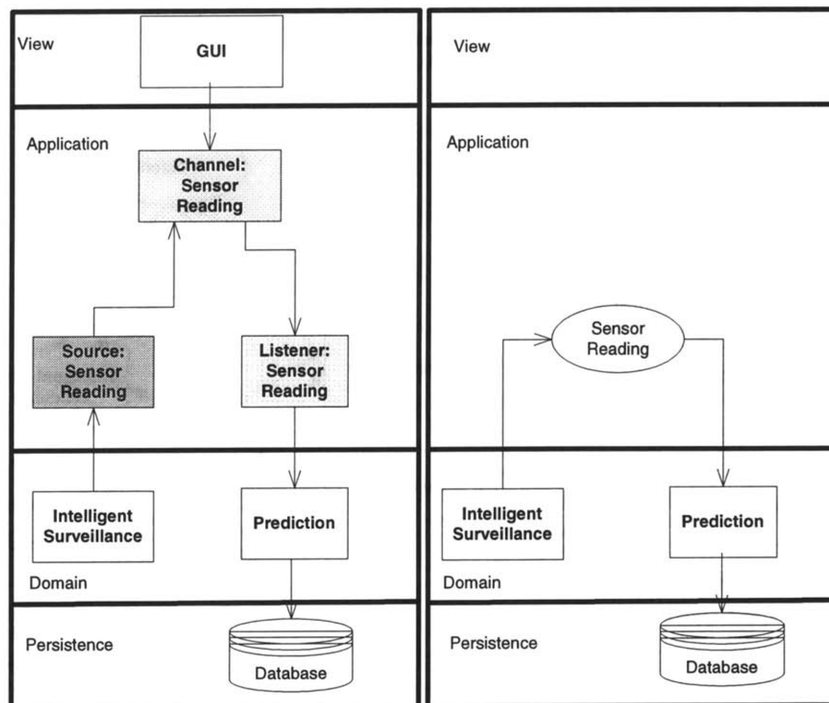


Figure 4-1: TMC design: connection of two systems

The right side of figure, shows the simplified representation, using arrows and ovals to represent the messages.

4.1.1 Centralized system, single region

An example of a design for a centralized system with a single network region is represented in Figure 4-2. In the figure, the subsystems are represented with rectangles. The Channels are represented with ovals and their type of message is written inside. Sources and Listeners have been omitted from the figure for the sake of simplicity, but every subsystem that produces a type message has a Source, and every subsystems that consumes a type message has a Listener.

The Intelligent Route Guidance and the Prediction subsystems form a cycle. The Intelligent Traffic Control and the Prediction subsystems form another cycle.

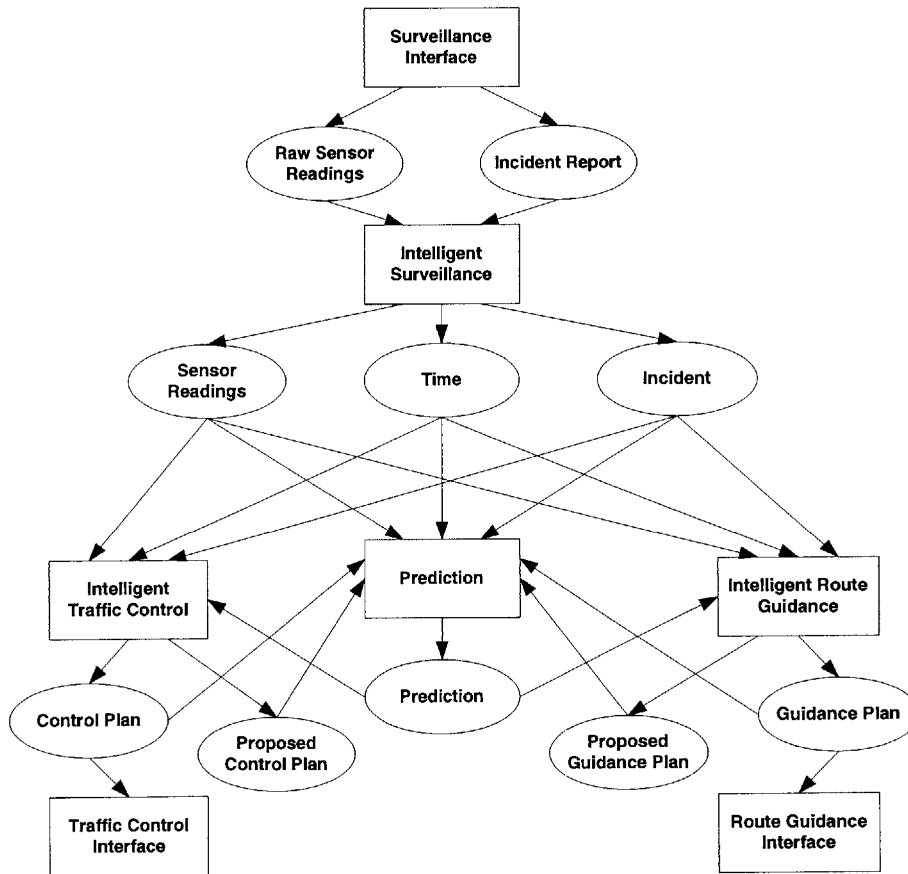


Figure 4-2: TMC design: centralized system, single region

4.1.2 Decentralized system, multiple regions

If the logic is decentralized, each region of the network works autonomously. The non-coordinated case, where the management operations over the regions are not interconnected consists of simply multiple instances of the system presented section 4.1.1. In a coordinated case, there exists a hierarchy among regions, as explained in Ben-Akiva et al. (1994).

Figure 4-3 presents the design for a decentralized hierarchical system operating over four regions under a coordinated strategy.

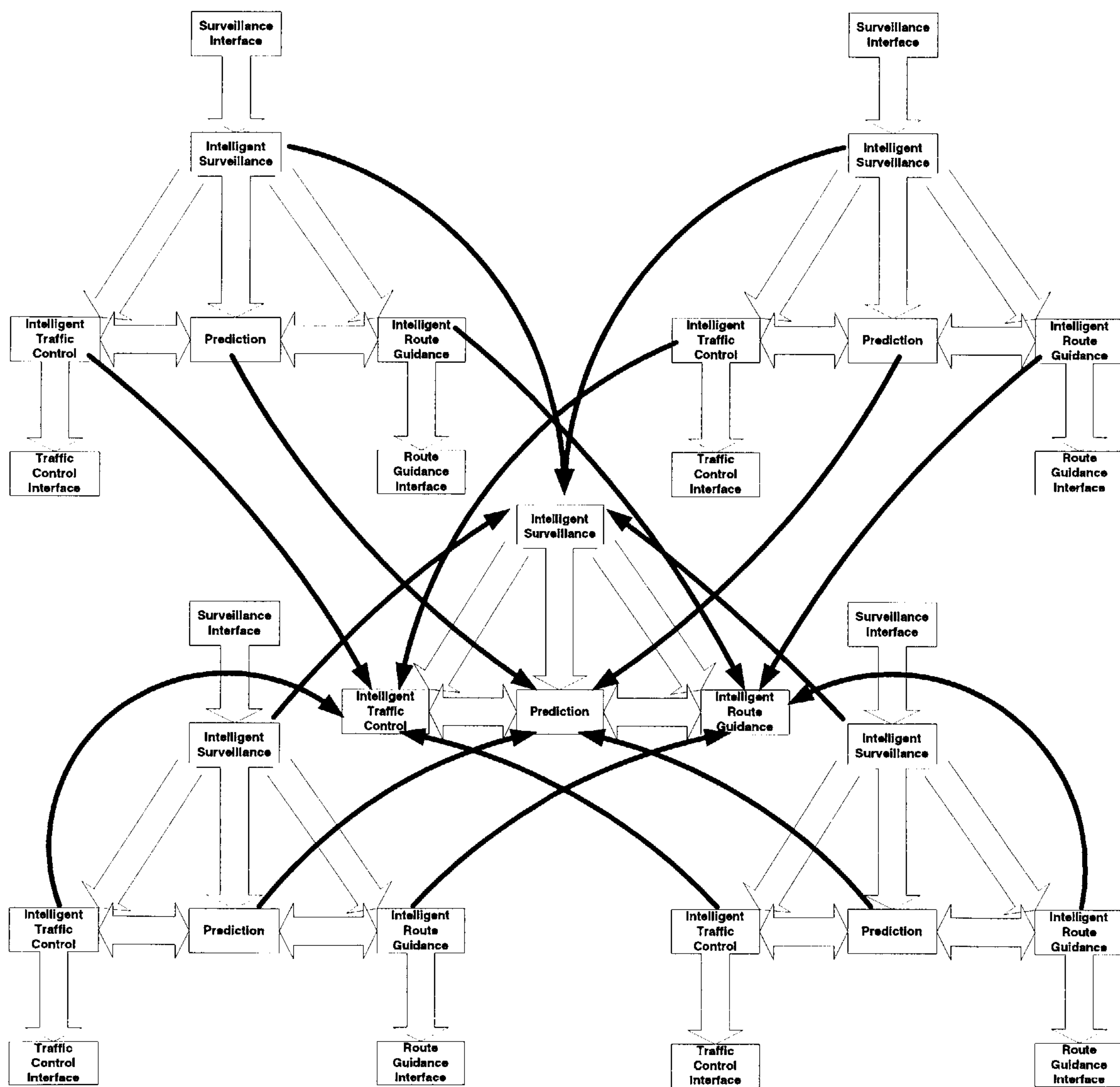


Figure 4-3: TMC design: decentralized system, multiple regions

4.2 Other applications of the proposed architecture

The proposed application architecture can also be used for the design of non-dynamic systems. These systems are special cases of the design proposed in 4.1.1. The static system corresponds to a system without prediction subsystem (and therefore no dynamic route guidance and dynamic traffic control are available). The off-line system is a simple case where the interfaces in the communication layer of the TMC (Surveillance, Traffic Control, and Route Guidance interfaces) are not connected to a real network but to a historical database. Finally, it is possible to use this architecture to propose a parallel traffic simulation system for multiple regions.

Static system

The proposed architecture can be used to integrate non dynamic ATMS/ATIS into a TMC. The system remains the same as in 4-2, but there is no prediction system. The static design of this system is represented in Figure 4-4. Because there is no prediction system, there is no iteration and no consistency check is needed.

Off-line system

An off-line version of a TMC is useful for DTMS evaluation purposes. The proposed architecture can be used to propose an off-line system design. The off-line system (see Figure 4-5) basically is the same as the 4.1.1, but the communication interfaces interact with databases instead of network devices. An off-line system uses historical traffic control plans and historical route guidance plans as an input, and therefore there is no consistency check.

Parallel simulation

A multiregional traffic simulation can be done using the proposed architecture. Each traffic simulator models the traffic conditions for one region, and a communication

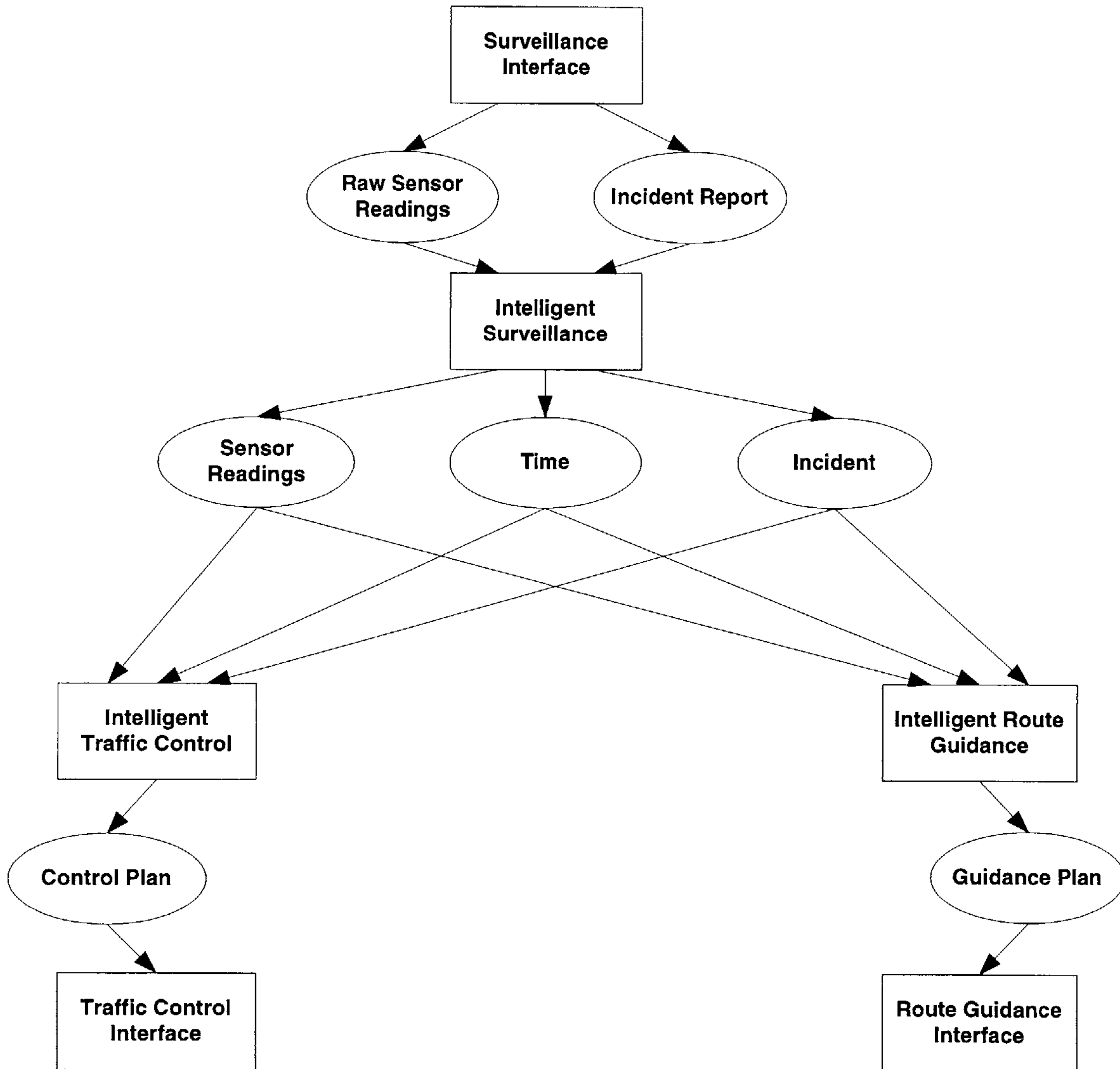


Figure 4-4: TMC design: static system with ATMS/ATIS

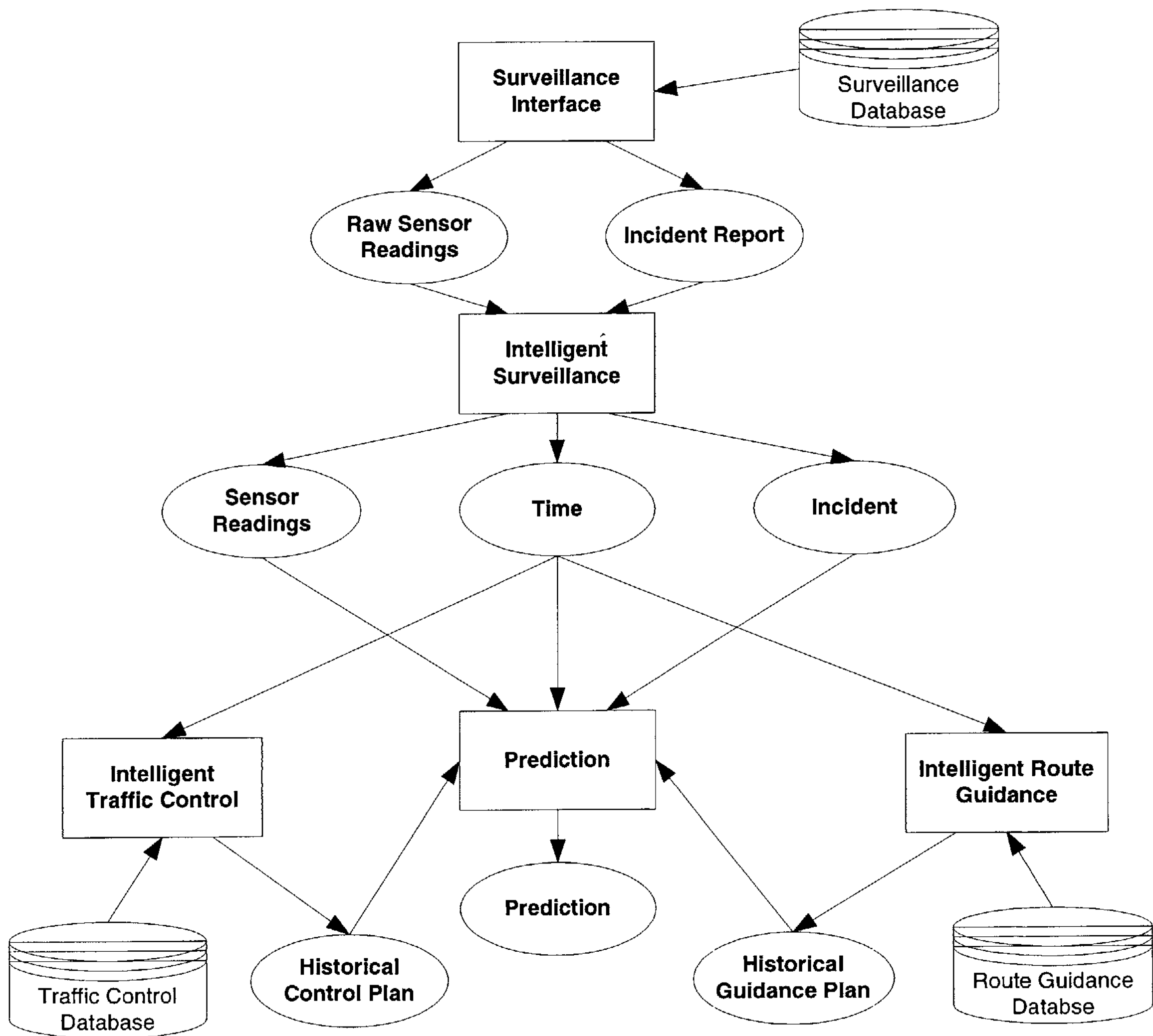


Figure 4-5: TMC design: off-line system for DTMS evaluation

system based on Sources, Channels and Listeners, ensures that surveillance from one region is received by the other regions. The communication can be done with only the surveillance interface level, or it can also include the route guidance and traffic control interfaces. In the last case, the operations of the network are coordinated. In Figure 4-6, a simple design of a system that allows the connection of four regions at the surveillance level is presented. The Channel is represented with an oval, and the Sources and the Listeners are represented with a small circle with a S and a L, respectively.

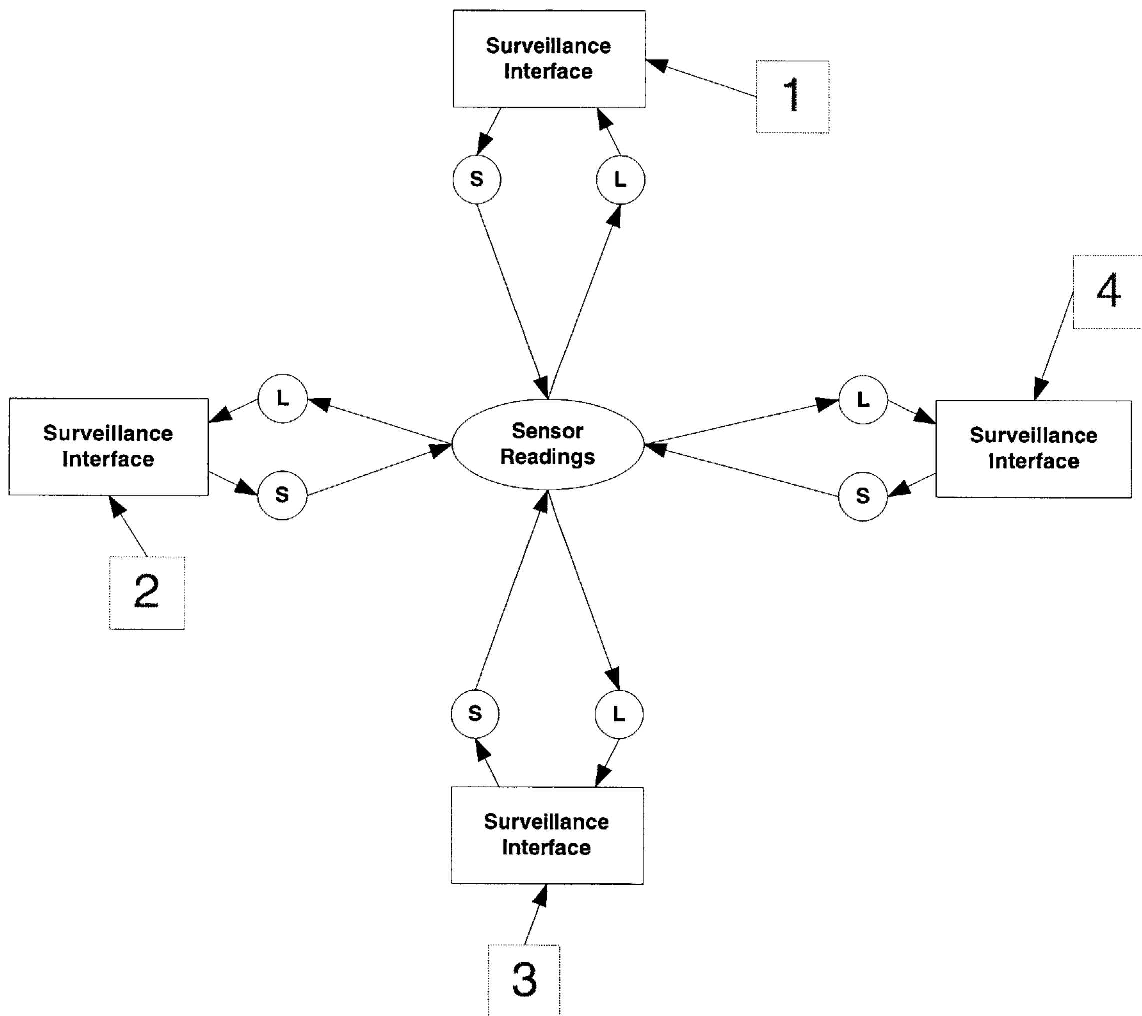


Figure 4-6: Multiregional simulation

4.3 Conclusion

We have seen in this chapter how the system architecture developed in chapter 3 does provide all the required functionality to design a TMC system with integrated DTMS. We can use the architecture to easily connect any two existing subsystems within the TMC, but we can also use to it to quickly integrate new subsystems, like an ATIS. Additionally, this architecture has proven useful for other applications, like parallel simulation. The four-layer-based TMC architecture proposed in 3.5.3 provides enough flexibility to be able to accommodate any DTMS within a any TMC.

Chapter 5

Case Study

The system architecture proposed in chapter 3 was used in chapter 4 to represent various TMC designs. In order to test the applicability of the architecture, we used it to develop a prototype TMC system that integrated one DTMS. The DTMS was DynaMIT, a prediction-based guidance system. We used MITSIMLab, a simulation laboratory for testing DTMS, to simulate the operations of the "real-world" and the TMC. However, the architecture is completely independent of whether we are integrating DynaMIT into a simulated TMC or a real TMC.

In the next section, we will review the requirements identified in chapter 2 with specific emphasis on MITSIMLab and DynaMIT. We then introduce MITSIMLab and DynaMIT, and we finish by analyzing the integrated system and the changes needed to go from two independent systems to an integrated simulated TMC.

5.1 Requirements

Let's recall some of the basic requirements we identified in previous chapters, and that are met by the proposed system architecture:

- *Open.* MITSIMLab and DynaMIT are complete different softwares. The integration of DynaMIT into MITSIMLab should not affect MITSIMLab nor DynaMIT's logic and internal implementation. Our goal is to adapt DynaMIT to MITSIMLab's design. This process should not be done by changing the

implementation and logic of MITSIMLab. Our goal is to use an open design, based on a plug-and-play architecture, that allows the integration of DynaMIT into MITSIMLab, but also into other simulators or TMCs.

- *Anonymity.* The only information that DynaMIT needs to know is that it is running inside of a TMC. Whether MITSIMLab or another simulator/TMC is performing those functions must be irrelevant for the design.
- *Parallelism.* In a simple case, like the one proposed here, one instance of DynaMIT is integrated into one instance of MITSIMLab. However, the design allows several instances of DynaMIT and MITSIMLab. In this case, the design needs to allow the parallel execution of all the instances.
- *Zero-Waiting time.* MITIMSLab cannot be blocked while DynaMIT generates a guidance. This suggests that the communication between DynaMIT and the MITSIMLab cannot be synchronous, but asynchronous.

5.2 MITSIMLab

MITSIMLab (Yang (1997)) consists of a traffic flow simulator (MITSIM) and a traffic management simulator (TMS). The traffic flow simulator models the driver behaviour and vehicular flow in the network, while the traffic management simulator mimics the control and routing functions.

The interaction (see Figure 5-1) between the traffic flows in the network and the control and route guidance generated by the system is a critical element for modelling dynamic traffic management systems. Traffic control and route guidance affects the behavior of individual drivers and, hence, traffic flow characteristics. The changes in traffic flows are in turn measured and potentially anticipated by the system and consequently influence present and future control and routing strategies.

Therefore, MITSIMLab can be understood as the interaction of the following subsystems:

- Traffic flow simulator (MITSIM).
- Traffic management center simulator (TMS).

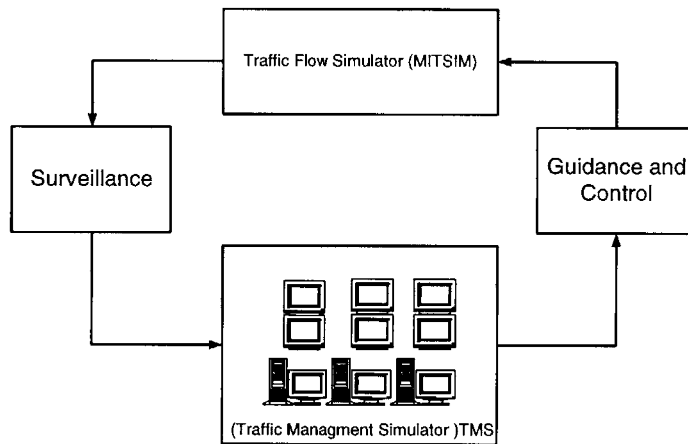


Figure 5-1: Structure of MITIMSLab

- Surveillance system.
- Control devices.
- Routing devices.

The control and route guidance provided by the traffic management system feed into the traffic flow simulator via various traffic control and routing devices in the simulated network. Drivers in the network respond to the traffic controls and guidance while interacting with one another. The outcome of the drivers' behavior is observed by the surveillance system module representing traffic sensors and probe vehicles. This module provides the traffic management simulator with the measurement of real-time traffic conditions.

Traffic Management Center Simulator (TMS)

TMS has a generic structure to model different types of systems. As discussed in section 1.4.2, control and route guidance systems can be classified as *pre-timed* and *adaptive* systems. In a pre-timed system, control and route guidance are predetermined based on historical traffic information using off-line analysis. In an adaptive system, control and route guidance are generated on-line based on real time traffic information obtained from surveillance sensors, and environmental conditions such as weather, scheduled construction work, etc.

Adaptive systems can be further divided into reactive and proactive systems. Most of the existing adaptive systems are reactive, whereas control and route guidance are provided based on prevailing traffic conditions. Proactive systems are the most recent development. A proactive system requires a prediction-based control and routing strategy. The structure of TMS can be represented as follows:

Route guidance

TMS broadcasts updated travel times to MITSIM's guidance system. MITSIM "transmits" the updated travel times to the vehicles equipped with on-board route guidance devices when vehicles enter the range of a communication beacon. Upon

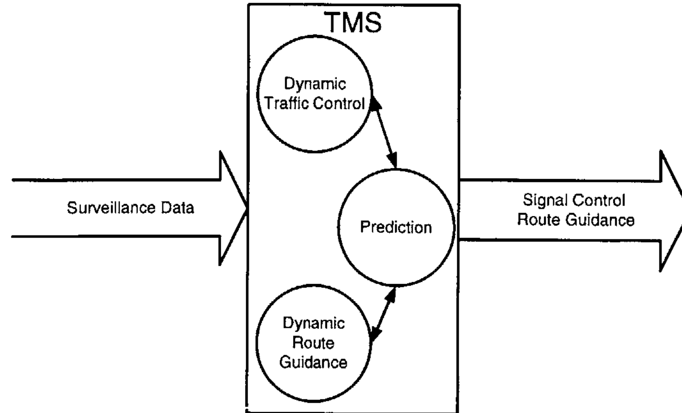


Figure 5-2: TMS: proactive traffic control and routing systems

receiving the new information, guided drivers select their routes based on the updated travel times. Any vehicle may respond to VMS according to a pre-specified compliance rate.

Traffic Control

The TMS module in MITSIMLab can simulate a wide range of traffic control and advisory devices. Examples include:

- *Intersection controls:* traffic signals, yield and stop signs.
- *Ramp controls:* ramp metering and speed limit signs.
- *Mainline controls:* lane use signs, variable speed limit signs , variable message signs, portal signals at tunnel entrances.

These signals and signs are controlled by four types of traffic signal controllers, namely *static*, *pre-timed*, *traffic adaptive*, and *metering* controllers. As the simulation proceeds a controller can switch from one type to another based on the implemented logic. For example, a controller may switch to traffic adaptive control in the off-peak period and to pre-timed control in the peak period.

Each controller is characterized by data items such as *controller type*, *signal type* (e.g., traffic signal at intersections, variable speed limit signs, lane use signs, etc.), number of *egresses* (i.e. the out-degree of the intersection node), *IDs of signals* it controls, timing table, etc.

5.3 DynaMIT

DynaMIT (Dynamic Network Assignment for the Management of Information to Travellers) is a real-time dynamic traffic assignment system. DynaMIT generates prediction-based guidance with respect to departure time, pre-trip path and mode choice decisions and en-route path choice decisions. In order to guarantee the credibility of the information system, the guidance provided by DynaMIT is consistent, meaning that it corresponds to traffic conditions that will most likely be experienced by drivers. Hence, DynaMIT provides user-optimal guidance, which implies that users cannot find a path that they would prefer compared to the one they chose based on the provided information. DynaMIT is organized around two main functions: state estimation, and prediction-based guidance generation. Figure 5-3 shows the overall structure and interactions among the various elements of DynaMIT. The state estimation component determines the current state of the network and demand levels given historical and surveillance data. Two simulation tools are being used iteratively in this context: the Demand Simulator and the Supply Simulator. The Demand Simulator estimates and predicts Origin-Destination (OD) flows and drivers' decisions in terms of departure time, mode and route choices. An initial estimate of the demand is directly derived from the data. The Supply Simulator explicitly simulates the interaction between that demand and the network. Assignment matrices are produced which map OD flows into link flows. The assignment matrices and real-time observations are then used by the Demand Simulator to obtain a better estimate of the demand. This loop is executed until congruity between demand and supply is obtained, at which point the simulation reproduces sufficiently well the observed data.

The prediction-based guidance generation module provides anticipatory guidance by using the network state estimate as input. Traffic prediction is performed for a given time horizon (e.g. thirty minutes). Guidance generation is based on an iterative process that evaluates traffic prediction and candidate guidance strategies. The system enforces consistency between the travel times on which the guidance

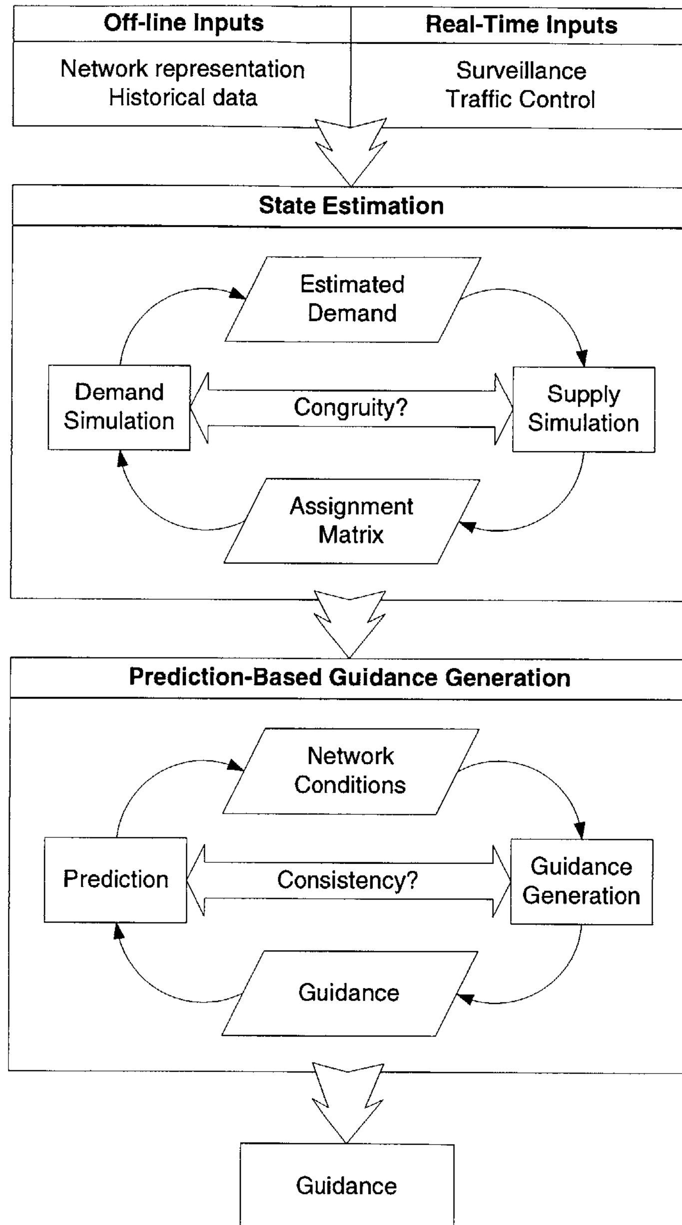


Figure 5-3: Structure of DynaMIT

is based and the travel times that result from travellers' reactions to the guidance. The quality of the prediction depends directly on the quality of the current state description. The state of the network is thus regularly estimated so that all available information is incorporated in a timely fashion and a new prediction can be computed.

We illustrate this concept with a simple example (see Figure 5-4). It is now 8:00 am. DynaMIT starts an execution cycle. It performs a state estimation using data collected during the last 5 minutes. When the state of the network at 8:00 am is available, DynaMIT starts predicting for a given horizon, say one hour, and computes a guidance strategy which is consistent with that prediction. At 8:07, DynaMIT has finished the computation, and is ready to implement the guidance strategy on the real network. This strategy will be in effect until a new strategy is generated. Immediately following that or at some pre-specified time, DynaMIT starts a new execution cycle. Now, the state estimation is performed for the last 7 minutes. While DynaMIT was busy computing and implementing the new guidance strategy, the surveillance system continued collecting real-time information, and DynaMIT will update its knowledge of current network conditions using that information. The new network estimate is used for a new prediction and guidance strategy. The process continues in a similar fashion during the entire time period of interest.

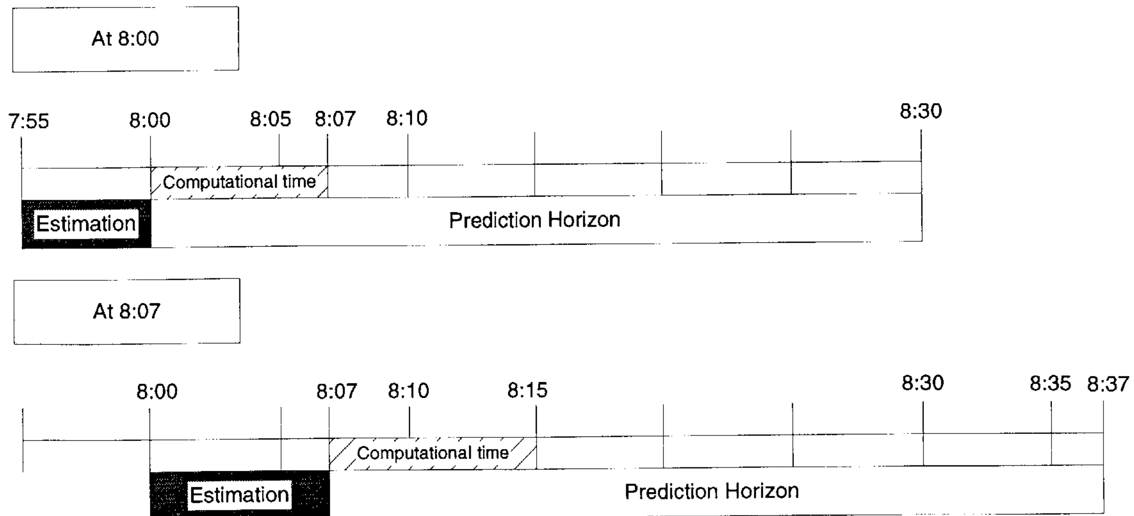


Figure 5-4: DynaMIT: Illustration of the rolling horizon

Intelligent Route Guidance in DynaMIT

The prediction-based guidance module consists of several interacting components:

- pre-trip demand simulation,
- OD flow prediction,
- network state prediction,
- guidance generation.

Aggregated historical demand is adjusted by the pre-trip demand simulator to account for departure time, mode, and route choices in response to guidance. This is used as input to the OD prediction model, which provides the required estimates of future OD flows. The network state prediction function undertakes the important task of traffic prediction for a given guidance strategy and predicted set of OD flows estimated by the state estimation module as a starting point. It uses a traffic simulation model and driver en-route behaviour models to predict the performance of the network for some time in the future. The guidance generation function uses the predicted traffic conditions to generate guidance according to the various ATIS in place. Note that for the current version, traffic control is loosely coupled with DynaMIT (i.e. control strategies are generated outside the DTA system, but possibly using inputs from it).

The guidance has to be consistent and unbiased. This means that there should be no better path than that taken by a driver in response to the provided information. In order to produce a guidance that satisfies these requirements, an iterative process is necessary. An iteration consists of a trial strategy, state prediction (network and demand) under the trial strategy, and evaluation of the predicted state (for consistency). In general, since the updated historical OD flows depend on future guidance and information, update of the historical OD flows (using the departure time and mode choice models) and OD prediction models are included in the iteration. This general case represents the situation where pre-trip guidance is available to the drivers. In the special case where only en-route guidance is available, the pre-trip

demand simulator is bypassed during the iterations. The initial strategy is generated from the prediction and guidance generation of the previous horizon.

Anticipatory descriptive guidance informs travelers about the traffic conditions they are likely to encounter on different feasible paths from their current position to their destination. Anticipatory prescriptive guidance recommends a path to travellers based on expected traffic conditions along alternative feasible paths. In both cases the guidance is called anticipatory because it is derived from forecasts of what link traffic conditions would be at the time that the links are actually traversed by a driver following the path. Note, however, that drivers who receive guidance information may change their paths as a result of that information. This may lead in turn to a change in the time-dependent link travel times on which the guidance information was based, and so the guidance may be invalidated. Guidance is called consistent if the traffic conditions which result from travelers' reactions to it are the same as those which were anticipated when generating it. Travelers receiving consistent descriptive guidance should experience traffic conditions which are the same as those which the guidance system predicted they would encounter, within the limits of modelling accuracy. In prescriptive guidance, consistency requires that the forecast traffic conditions which are the basis of the guidance recommendations reflect (again within the limits of modelling accuracy) the conditions which actually prevail on the network as a result of informed drivers reacting to the guidance.

5.4 Implementation of the integration of DynaMIT in MITSIMLab

For the purposes of this work, MITSIMLab plays the role of the "real-world" and the TMC. In this section, we illustrate how DynaMIT is integrated into MITSIMLab using the architecture developed in chapter 3.

The TMC design proposed in section 4.1.1 was used as a reference to create the design of the integrated MITSIMLab-DynaMIT system. DynaMIT was placed

into MITSIMLab's TMS using a series of adaptors[1]. All these adaptors where developed and located in a separate process within MITSIMLab called TMCA (Traffic Management Center Adaptors). TMCA provides access to TMS' public subsystems without modifying the software implementation of TMS. In the Communication Layer of TMS (see section 2.1 for a description of the Communication Layer), TMCA provides adaptors to TMS' Surveillance Interface, Traffic Control Interface and Route Guidance Interface. In the Logic Layer of TMS, TMCA provides an adaptor to the results of the current control plan.

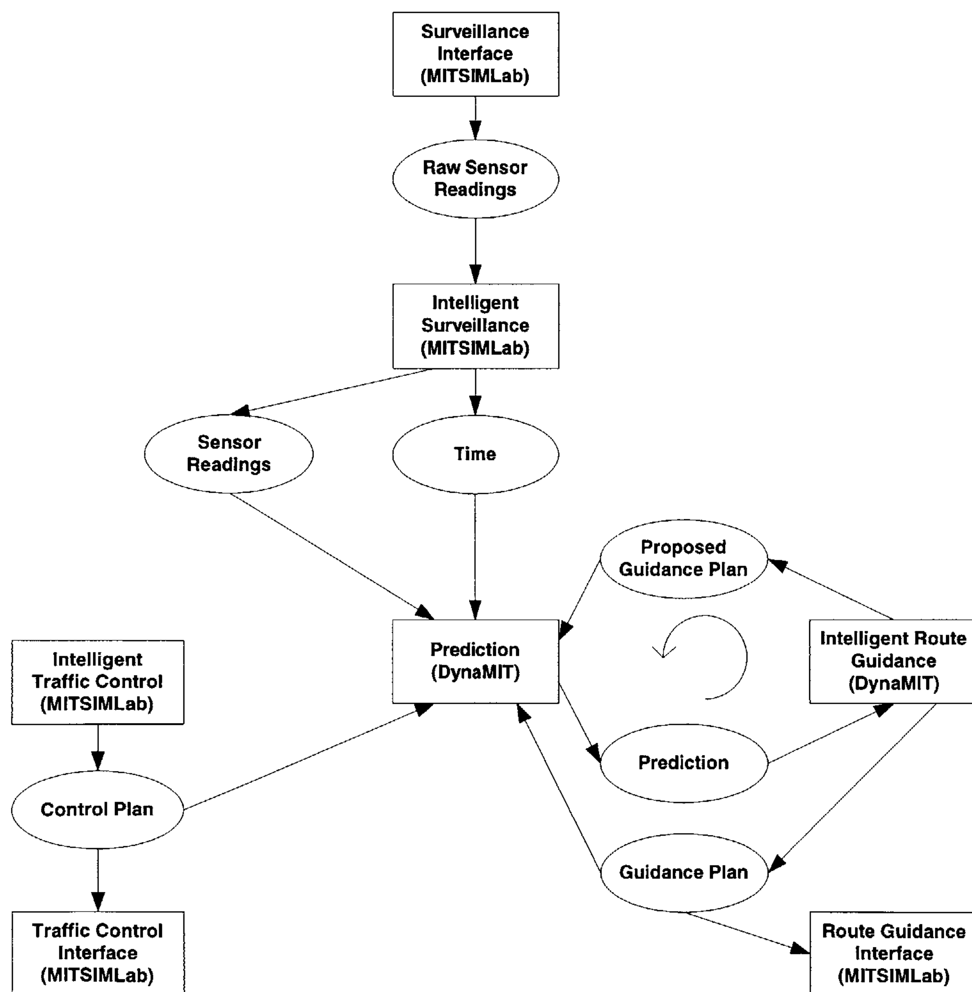


Figure 5-5: Integration of DynaMIT into MITSIMLab: system design

Figure 5-5 represents the system design of the integrated system with DynaMIT

[1]For a description of the adaptor pattern, see Gamma et al. (1994).

working within TMS. This design has been implemented in C++. The benefits of using the architecture proposed in chapter 3 were evident when programming the adaptors to MITSIMLab and DynaMIT.

The objects that constitute the communication system (Sources, Channels, Listeners, Registry, etc.) are generic and did not need to be changed to implement the integration of DynaMIT into MITSIMLab. Our efforts consisted of implementing the messages that are exchanged among the systems (time, sensor readings, control plan, and guidance plan), and implementing the adaptors in both MITSIMLab and DynaMIT.

Appendix B contains a detailed description of the communication objects and appendix C contains a detailed description of the particular messages created for the MITSIMLab/DynaMIT integration. Most of these messages can probably be reused in other system integrations.

As shown in Figure 5-5, surveillance data is transmitted from MITSIM (simulating the "real-world) to TMS (simulating the TMC) at periodic aggregation intervals (e.g. every minute). TMS translates the data received from MITSIM into data that can be used for all subsystems residing within TMS (or the TMC). TMCA is the part of TMS responsible for translating the data received from MITSIM into a generic format understood by all subsystems.

TMCA process starts with an initialization stage. In this stage, TMCA begins by finding the Registry:

```
// bind to the Registry
// NOTE: this is the only Orbix specific code (_bind)
// and this is why we need a marker

theRegistry =  DTMS_Registry::_bind( theFileManager->registryMarker_ ,
                                     theFileManager->registryHostName_ );
```

it then looks in the Registry for the location of the Factories, and binds to it:

```
DTMS_Name aName;
aName.marker = CORBA::string_alloc(64);
strcpy( aName.marker, theFileManager->factoryName_ );
aName.adaptorType = DTMS_TYPE_FACTORY;

theFactory =  DTMS_Factory::_narrow( theRegistry->resolve( aName ) );

// find our MessageFactories (one for each type of source)

strcpy( aName.marker, theFileManager->timeMessageFactoryName_ );
aName.adaptorType = DTMS_TYPE_MESSAGEFACTORY;

theTimeMessageFactory = DTMS_MessageFactory::_narrow(
          theRegistry->resolve( aName ) );

strcpy( aName.marker, theFileManager->sensorReadingMessageFactoryName_ );
aName.adaptorType = DTMS_TYPE_MESSAGEFACTORY;

theSensorReadingMessageFactory = DTMS_MessageFactory::_narrow(
                              theRegistry->resolve( aName ) );
```

finally, we request theFactory to create as many sources and listeners as we need:

```
theTimeSource = theFactory->createSource( theFileManager->timeSourceName_,
                                          DTMS_TYPE_TIME );

theSensorReadingSource = theFactory->createSource(
                        theFileManager->sensorReadingSourceName_,
                        DTMS_TYPE_SENSOR_READING );

theGuidanceListener = theFactory->createListener(
                       theFileManager->guidanceListenerName_,
                       DTMS_TYPE_GUIDANCE );
```

At this stage, TMCA appears in the Registry as a source of surveillance data and a listener of guidance.

The user can browse all of the entries in the Registry in a User Interface and assign Sources to Listeners. The User Interface uses one or multiple Factories to create as many Channels as are needed to connect the Sources with the Listeners. This process is completely transparent to the user.

Every time TMS has new surveillance data, TMCA broadcasts every sensor reading to all the connected listeners. The following code broadcasts one sensor reading:

```
DTMS_SensorReadingInfo info;
 info.sensorID   = sensor;
 info.count      = count;
 info.speed      = speed;
 info.occupancy  = occupancy;
 info.fromTime   = fromTime;
 info.toTime     = toTime;

DTMS_SensorReadingMessage_ptr pMessage =
   DTMS_SensorReadingMessage::_narrow(
      theSensorReadingMessageFactory->createMessage() );

pMessage->setValue( info );

theSensorReadingSource->notify( pMessage );
```

Similarly, DynaMIT also has sources and listeners. The listeners receive time information, sensor readings, and control settings. One source produces a guidance (travel times). This guidance is received by TMCA. The following code illustrates this process:

```
DTMS_GuidanceInfo info;
```

```
DTMS_GuidanceMessage_ptr ptrGuidanceMessage = DTMS_GuidanceMessage::_narrow(
                              theGuidanceListener->getNextMessage() );

info = ptrGuidanceMessage->getValue();

theGuidanceListener->clearBuffer();
```

Chapter 6

Conclusions

6.1 Research Contribution

This research contributes to: (a) the development a generic system architecture that permits the integration and parallelization of multiple legacy systems under a cooperative framework; (b) the analysis and design of different integrated Traffic Management Centers systems based on this system architecture; (c) the demonstration of the system architecture by developing an integrated Traffic Management Center based on MITSIMLab using DynaMIT.

The system architecture is the result of the integration of different designs patterns. To provide distribution mechanisms, a layer on top of the Common ORB Architecture was built. CORBA basic synchronous communication paradigm was extended using the Publisher/Subscriber pattern (*push model*). A modified version of this pattern constitutes the core of the system architecture and provides parallelization mechanisms to the architecture. This pattern is based on the collaboration of three objects: the Source, that multicasts multiple types of information; the Listeners, that consume different types of information; and the Channels, an intermediate layer to isolate the communication between from Sources to Listeners. The parallelization mechanisms are supported with a naming service (Registry) that provides a unique location mechanism for all the systems in the architecture independently of the particular Object Request Broker (ORB) used to

create the objects. Finally, a creation mechanism, based on the Abstract Factory pattern, was developed to isolate concrete object instantiation, dependent on the ORB internal mechanisms, into abstract objects.

The capabilities of the proposed system architecture were demonstrated through the design of several systems that integrated different DTMS into TMCs. The design of a TMC system that controls a single region with dynamic traffic control, dynamic route guidance, and dynamic traffic assignment subsystems was generated using the system architecture. Other systems with different features, such as decentralized control, multiple region control, and off-line evaluation, were also generated using the same system architecture.

The applicability of the system architecture was proved by implementing one of the system designs. The case study showed the integration of DynaMIT, a DTMS with anticipatory travel information and route guidance based on prediction, into a Traffic Management Simulator (TMS), part of MITSIMLab, a traffic simulation laboratory for DTMS evaluation. The case study showed that the neither system (DynaMIT nor TMS) needed to be changed in order to comply with the proposed architecture. The implementation consisted of a series of adaptors that were developed in order to interface both systems with the core of the architecture.

In conclusion, the work of this research can be used to integrate DTMS into Traffic Management Centers.

6.2 Future Work

The work of this research can be extended in the following directions:

Traffic Management Systems Design

New Traffic Management Centers systems and reengineered systems can be based on this system architecture. New TMCs built on top of this architecture provide a plug-and-play environment that allows the easy integration of various Dynamic Traffic Management Systems. The TMC systems that can be designed based on this

architecture can be extremely complex, and the interactions between systems reach levels that could not be thought in the past. Research in the possibilities that this architecture provides in terms of TMC design could be followed.

Artificial Intelligence Systems

A TMC built under this architecture can have multiple Route Guidance and Traffic Control systems, and an intelligent decision module can be in charge of dynamically selecting the adequate system for each traffic situation. Further work is necessary to design this decision module and its integration in this architecture (it basically substitutes the decisions taken by an operator).

Dynamic Traffic Control

This work of this research focused more on the design of a system based on this architecture that allows the integration of DTMS with anticipatory route guidance into TMCs. As such, the case study was based on DynaMIT, a prediction-based guidance system. Research to integrate dynamic traffic control systems in TMCs based in this architecture could also be done. This implies defining the structure of the Messages, and establishing Listeners, Sources and Channels that can transmit these Messages.

Microsimulation parallelization

One of the most innovative applications of this system architecture was introduced in section 4.2. This architecture provides the core components needed to integrate several parallel systems. As such, a large network with arterials, freeways and urban ways can be divided into small regions. Each region can be modelled by a different microscopic traffic simulator running with different time steps. Every time a vehicle gets to the boundaries of one the regions modelled by a simulator, it is broadcasted to the other systems using a Listener/Channel/Source design. Another simulator will use this information to enter that vehicle in its network region. A time event is

necessary to synchronize the execution of all the regions. All the simulators broadcast their own simulation time and receive all multiple simulation times from the other systems. At any instant the current time is the minimum of these time signatures. Fast simulations will need to wait for slow ones.

Bibliography

P. Aicher, F. Busch, and R. Gloger. Use of route guidance data for better traffic control and data management in integrated systems. In *Proceedings of the 9th DRIVE Conference*, Brussels, 1991.

M. E. Ben-Akiva, M. Bierlaire, J. Bottom, H. N. Koutsopoulos, and R. G. Mishalani. Development of a route guidance generation system for real-time application, 1997. Presented at 8th IFAC Symposium on Transportation Systems.

Moshe E. Ben-Akiva, Haris N. Koutsopoulos, and Anil Mukandan. A dynamic traffic model system for ATMS/ATIS operations. *IVHS Journal*, 1(4), 1994.

J. Bottom, M. Ben-Akiva, M. Bierlaire, and Chabini I. Generation of consistent anticipatory route guidance, 1998. Presented at TRISTAN III Symposium.

Kyle Brown and Bruce Whitenack. Crossing chasms: The static patterns. *Pattern Languages of Program Design*, II, 1996.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons, 1996.

Owen J. Chen. A dynamic traffic control model for real time freeway operations. Master's thesis, Massachusetts Inst. of Tech., Cambridge, MA, 1996.

Owen J. Chen. *Integration of Dynamic Traffic Control and Assignment.* PhD thesis, Massachusetts Inst. of Tech., Cambridge, MA, 1998.

DYNA. DYNA – A dynamic traffic model for real-time applications – DRIVE II project. Annual review reports and deliverables, Commision of the European Communities - R&D programme telematics system in the area of transport, 1992-1995.

FHWA. DTA RFP. Technical report, Federal Highway Administration, US-DOT, McLean, Virginia, 1995.

Loral AeroSys FHWA. Traffic management center - the state-of-the-practice. Task A: Final Working Paper for Design of Support Systems for ATMS DTFH61-92C-00073, U.S. Department of Transportation - Federal Highway Administration, 1993.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

Nathan H. Gartner and Chronis Stamatiadis. Integration of dynamic traffic assignment with real time traffic adaptive control. 76th Transportation Research Board Annual Meeting, 1997.

Nathan H. Gartner, Chronis Stamatiadis, and Philip J. Tarnoff. Development of advanced traffic signal control strategies for IVHS: A multi-level design. *Transportation Research Record*, 1494, 1995.

Hague Consulting Group. What happens: European trials of anticipatory traffic control. *Traffic Technology International*, 8:64–68, 1997.

IONA Technologies Ltd. IONA. *Orbix 2: distributed object technology.* IONA Technologies Ltd., Cambridge, MA 02139, 1995.

I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, ACM Press, Reading, Massachusetts, 1992.

Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, ACM Pres, Reading, Massachusetts, 1998.

Hani S. Mahmassani, Ta-Yin Hu, Sriniva Peeta, and Athanasios Ziliaskopoulos. Development and testing of dynamic traffic assignment and simulation procedures for ATIS/ATMS applications. Report DTFH61-90-R-00074-FG, U.S. DOT, Federal Highway Administration, McLean, Virgina, 1994.

MIT. Development of a deployable real-time dynamic traffic assignment system. Technical Report Task B-C, Massachusetts Inst. of Tech., Intelligent Transportation Systems Program and Center for Transportation Studies, Cambridge, MA, 1996. Interim reports submitted to the Oak Ridge National Laboratory.

Object Management Group OMG. *The Common Object Request Broker: Architecture and Specification.* http://www.omg.org, Framingham, MA, 2.0 edition, 1995.

Object Management Group OMG. *OMG Unified Modelling Language Specification.* http://www.omg.org, Framingham, MA, 1998.

Object Management Group OMG. *The Common Object Request Broker: Architecture and Specification.* http://www.omg.org, Framingham, MA, 2.3 edition, 1999.

Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA.* Wiley, John & Sons, February 1998, 2nd edition.

Christof Peter and Franz Puntigam. Static type checking and deadlock prevention in systems based on asynchronous message passing. In *Workshop on Models, Formalisms and Methods for Object-Oriented Distributed Computing*, Jyväskylä, Finland, 1997.

Kay Römer and Arno Puder. *MICO: CORBA 2.3 Implementation*. Universität Frankfurt, Frankfurt am Main, Germany, 1999.

John Schettino and Liz O'Hara. *CORBA for dummies*. International Data Group, IDG Books Worldwide, 1998.

Douglas C. Schmidt and Steve Vinoski. Object interactions: Developing c++ servant classes using the portable object adapter (column 13). *C++ Report*, 1998.

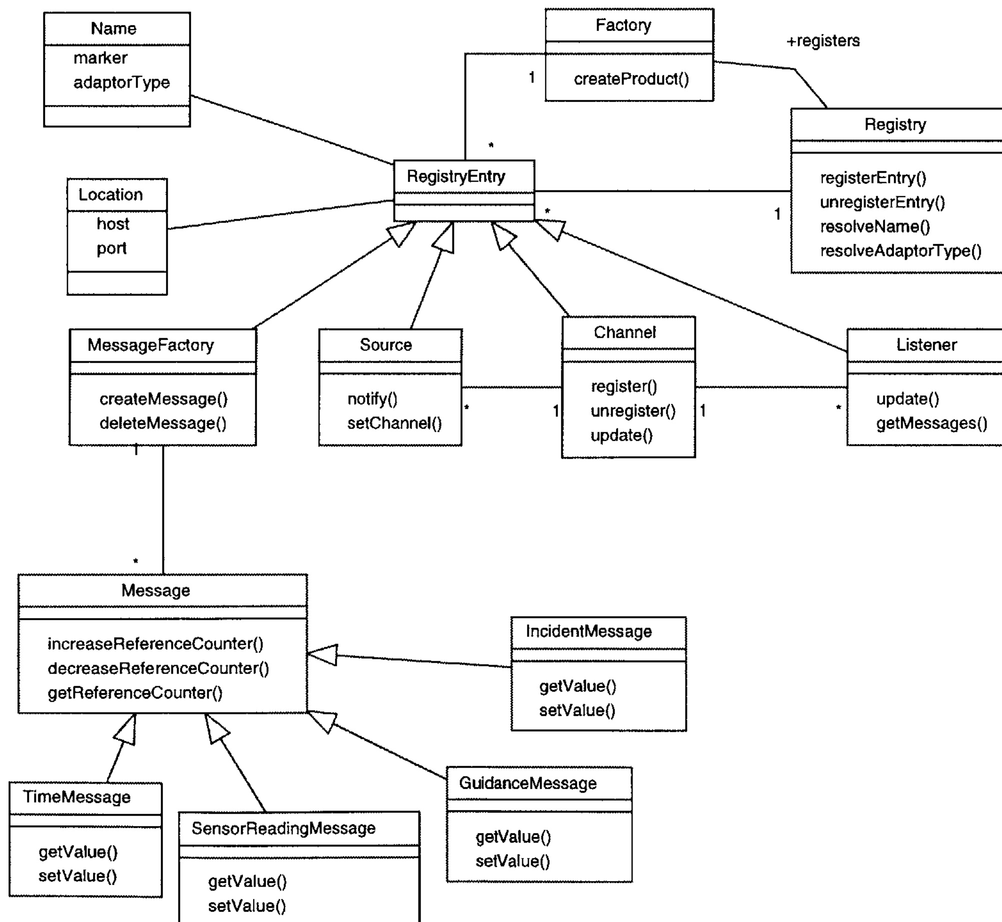
Andrew Tanenbaum. *Computer Networks*. Prentice-Hall, second edition, 1988.

Qi Yang. *A Simulation Laboratory for Evaluation of Dynamic Traffic Management Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1997.

Ed. Yourdon, K. Whitehead, J. Thomann, K. Oppel, and P. Nevermann. *Mainstream objects: an analysis and design approach for business*. Prentice Hall, Upper Saddle River, New Jersey, 1995.

Appendix A

UML Object diagram



Appendix B

Object definition files

— Start of DTMSObject.idl —

```
/* DTMS_Object.idl
 *
 * 11/16/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_Object_idl
#define DTMS_Object_idl

/**
 * The basic DTMS_Object
 * All objects in DTMS must inherit from this object
 */
interface DTMS_Object
{
};

#endif
```

— End of DTMSObject.idl —

— Start of DTMSTypedefs.idl —

```
/* DTMS_TypeDefs.idl
 *
 * 11/19/99 Bruno M Fernandez Ruiz
 *
 */

#ifndef DTMS_Typedef_idl
#define DTMS_Typedef_idl

/** DTMS_Time are seconds since GMT 00:00:00 01/01/1970
 */
typedef double DTMS_Time;

/** A Time interval
 */
struct DTMS_TimeInterval
{
  DTMS_Time fromTime;
  DTMS_Time toTime;
};

/**
 * The type of adaptors existing in the DTMS system
 */
enum DTMS_AdaptorType
{
  DTMS_TYPE_CHANNEL,
  DTMS_TYPE_SOURCE,
  DTMS_TYPE_LISTENER,
  DTMS_TYPE_FACTORY,
  DTMS_TYPE_MESSAGEFACTORY
};
```

```
enum DTMS_AdaptorSubtype
{
  DTMS_TYPE_NULL,
  DTMS_TYPE_TIME,
  DTMS_TYPE_SENSOR_READING,
  DTMS_TYPE_GUIDANCE
};


#endif
```

— End of DTMSTypedefs.idl —

— Start of DTMSRegistry.idl —

```
/* DTMS_Registry.idl
 *
 * 11/16/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_Registry_idl
#define DTMS_Registry_idl
#include <DTMS_Object.idl>
#include <DTMS_Name.idl>
#include <DTMS_Locator.idl>
#include <DTMS_RegistryEntry.idl>

/** The DTMS_Registry object
 * The DTMS_Registry object is responsible for keeping an up to date
 * database with all servers within the DTMS.
 * The DTMS_Registry can register, resolve, and search for  a name
 * within a type.
```

```
 * @see DTMS_Name, DTMS_RegistryEntry
 */
interface DTMS_Registry : DTMS_Object
{
  /** Registers a name at a certain host
   */
  oneway void registerEntry(in DTMS_RegistryEntry entry );
  /** Unregister a name
   * @returns true if successful
   */
  boolean unregisterEntry(in DTMS_RegistryEntry entry );
  /** Resolve in which host and port the server with that name
   * is running
   * @returns An structure DTMS_Locator with the name of the host
   */
  DTMS_Locator resolveName(in DTMS_Name name);
  /** Get a reference to that name to be used in a narrow in
   * a client
   * @returns the entry
   */
  DTMS_RegistryEntry resolve(in DTMS_Name name);
  /** Searchs the registry for names that implement a certain
   * adaptor type
   * @returns A list of the entries that implement that type
   */
  DTMS_RegistryEntryList resolveAdaptorType( in DTMS_AdaptorType type);
  /**
   */
  void checkConsistency();
};

#endif
```

— End of DTMSRegistry.idl —

— Start of DTMSRegistryEntry.idl —

```
/* DTMS_RegistryEntry.idl
 *
 * 11/16/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_RegistryEntry_idl
#define DTMS_RegistryEntry_idl
#include <DTMS_Object.idl>
#include <DTMS_Name.idl>
#include <DTMS_Locator.idl>

/** The DTMS_RegistryEntry object
 * Every object that wants to be inserted in the
 * DTMS_Registry must implement this interface
 * @see DTMS_Name, DTMS_Locator, DTMS_Registry
 */
interface DTMS_RegistryEntry : DTMS_Object
{
  /** The name of the entry
   */
  readonly attribute DTMS_Name name;
  /** The locator of the entry
   */
  readonly attribute DTMS_Locator locator;
  /** Dummy method
   */
  oneway void ping();
};
```

```
typedef sequence<DTMS_RegistryEntry> DTMS_RegistryEntryList;

#endif
```

— End of DTMSRegistryEntry.idl —

— Start of DTMSName.idl —

```
/* DTMS_Name.idl
 *
 * 12/16/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_Name_idl
#define DTMS_Name_idl
#include <DTMS_Typedefs.idl>

/**
 * A name is based on the implementation's object name and the type
 * of adaptor it implements.
 */
struct DTMS_Name
{
  /** The name of the object that implements a type of adaptor
   */
  string marker;
  /** The type of adaptor being implemented
   */
  DTMS_AdaptorType adaptorType;
  /**
   */
  DTMS_AdaptorSubtype adaptorSubtype;
};
```

```
/**
 */
typedef sequence<DTMS_Name> DTMS_NameList;



#endif
```

— End of DTMSName.idl —

— Start of DTMSLocator.idl —

```
/* DTMS_Locator.idl
 *
 * 12/16/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_Locator_idl
#define DTMS_Locator_idl
#include <DTMS_Typedefs.idl>


/**
 * A locator permits to describe the location of a server in
 * more or less detail depending on the implementation of CORBA.
 */
struct DTMS_Locator
{
  /** The host name, either IP or real name
   */
  string host;
  /** The port that listens to requests for this server
   * With Orbix this is set using an environment variable when
   * launching the orbix daemon in mode unregistered (orbixd -u)
```

```
   * With MICO this is set either in a configuration file or as a
   * paramenter to the ORB adaptor.
   * IDL: 16 bits
   */
  unsigned short portNumber;
};

/**
 */
typedef sequence<DTMS_Locator> DTMS_LocatorList;

#endif
```

— End of DTMSLocator.idl —

— Start of DTMSFactory.idl —

```
/* DTMS_Factory.idl
 *
 * 11/16/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_Factory_idl
#define DTMS_Factory_idl
#include <DTMS_Object.idl>
#include <DTMS_Typedefs.idl>
#include <DTMS_RegistryEntry.idl>
#include <DTMS_Channel.idl>
#include <DTMS_Listener.idl>
#include <DTMS_Source.idl>

/** The DTMS_Factory object
 * The DTMS_Factory object is responsible for creating the
```

```
 * channels, listeners, and sources  for the different
 * types of messages.
 * @see DTMS_Channel, DTMS_Message, DTMS_RegistryEntry
 */
interface DTMS_Factory : DTMS_RegistryEntry
{
  /** Create a channel in this factory
   * and register it in the registry with
   * this name and adaptortype
   * @returns The created channel
   */
  DTMS_Channel createChannel( in string channelName,
                      in DTMS_AdaptorSubtype channelType );
  /** Delete a channel
   */
  void deleteChannel( in DTMS_Channel channel );
  /** Create a listener in this factory
   * and register it in the registry with
   * this name and adaptortype
   * @returns The created channel
   */
  DTMS_Listener createListener( in string listenerName,
in DTMS_AdaptorSubtype listenerType );
  /** Delete a listener
   */
  void deleteListener( in DTMS_Listener listener );
  /** Create a source in this factory
   * and register it in the registry with
   * this name and adaptortype
   * @returns The created channel
   */
  DTMS_Source createSource( in string sourceName,
```

```
    in DTMS_AdaptorSubtype sourceType );
  /** Delete a source
   */
  void deleteSource( in DTMS_Source source );
};

#endif
```

— End of DTMSFactory.idl —

— Start of DTMSSource.idl —

```
/* DTMS_Source.idl
 *
 * 12/22/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_Source_idl
#define DTMS_Source_idl
#include <DTMS_Typedefs.idl>
#include <DTMS_RegistryEntry.idl>
#include <DTMS_Channel.idl>


/** The DTMS_Source object
 * Any object that wants to broadcast a messagge to
 * any channel must be a source and implement this interface.
 * @see DTMS_Listener, DTMS_Channel
 */
interface DTMS_Source : DTMS_RegistryEntry
{
  /** set the channel
   */
```

```
  void setChannel( in DTMS_Channel aChannel );
  /** Notify the channel
   */
  oneway void notify( in DTMS_Message aMessage );
};

typedef sequence<DTMS_Source> DTMS_SourceList;

#endif
```

— End of DTMSSource.idl —

— Start of DTMSChannel.idl —

```
/* DTMS_Channel.idl
 *
 * 12/22/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_Channel_idl
#define DTMS_Channel_idl
#include <DTMS_Typedefs.idl>
#include <DTMS_RegistryEntry.idl>
#include <DTMS_Listener.idl>
#include <DTMS_Message.idl>

interface DTMS_Source;

/** The DTMS_Channel object
 * The DTMS_Channel object is responsible for broadcasting the current
 * messages received from the sources to all the listeners.
 * @see DTMS_Source, DTMS_Listener
 */
```

```
interface DTMS_Channel : DTMS_RegistryEntry
{
  void registerListener(in DTMS_Listener listener );
  /** Unregister a listener
   * @returns true if successful
   */
  boolean unregisterListener(in DTMS_Listener listener );
  /** Notify the listeners and increase the reference
   * count of the message
   */
  oneway void notify( in DTMS_Message time );
  /** This operation is called by the listeners
   * everytime their clearBuffer() operations is called.
   * It decreases the reference count of the message
   */
  oneway void clearBuffer( in DTMS_Message message );

  /** Check that all the listeners are alive
   */
  void checkConsistency( );
};

typedef sequence<DTMS_Channel> DTMS_ChannelList;


#endif
```

— End of DTMSChannel.idl —

— Start of DTMSListener.idl —

```
/* DTMS_Listener.idl
 *
```

```
 * 12/22/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_Listener_idl
#define DTMS_Listener_idl
#include <DTMS_Typedefs.idl>
#include <DTMS_RegistryEntry.idl>
#include <DTMS_Message.idl>



interface DTMS_Channel;
interface DTMS_Listener;

typedef sequence<DTMS_Listener> DTMS_ListenerList;

/** The DTMS_Listener object
 * Any object that wants to receive any message broadcast from any
 * channel must be a listener and implement this interface.
 * @see DTMS_Source, DTMS_Channel
 */
interface DTMS_Listener : DTMS_RegistryEntry
{
  /** Update the time
   */
  oneway void update( in DTMS_Message theTime );
  /**
   * @returns the next message in the queue
   */
  DTMS_Message getNextMessage( );
  /**
   * @returns the stored messages
   */
```

```
  DTMS_MessageList getMessages( );
  /** Clear the list of messages
   */
  oneway void clearBuffer();
  /** Set the channel
   */
  void setChannel( in DTMS_Channel theChannel );
};



#endif
```

— End of DTMSListener.idl —

— Start of DTMSMessageFactory.idl —

```
/* DTMS_MessageFactory.idl
 *
 * 11/16/99 Bruno M Fernandez Ruiz
 *
 */
#ifndef DTMS_MessageFactory_idl
#define DTMS_MessageFactory_idl
#include <DTMS_Object.idl>
#include <DTMS_Typedefs.idl>
#include <DTMS_RegistryEntry.idl>
#include <DTMS_Message.idl>

/** The DTMS_MessageFactory object
 * The DTMS_MessageFactory object is an abstract object
 * responsible for creating themessages. Every concrete
 * factory must inherit from this object.
```

```
 * THIS IS AN ABSTRACT FACTORY.
 * @see DTMS_Message
 */
interface DTMS_MessageFactory : DTMS_RegistryEntry
{
  /** Create a message
   * @returns The created message
   */
  DTMS_Message createMessage();
  void deleteMessage( in DTMS_Message message );
};

#endif
```

— End of DTMSMessageFactory.idl —

Appendix C

Message definition files

— Start of DTMSMessage.idl —

```
/* DTMS_Message.idl
 *
 * 12/22/99 Bruno M Fernandez Ruiz
 */
#ifndef DTMS_Message_idl
#define DTMS_Message_idl
#include <DTMS_Typedefs.idl>
#include <DTMS_Object.idl>

/** The DTMS_Message object
 * The DTMS_Message is the base class for all the messages that can
 * be broadcasted in the Source/Channel/Listener chain.
 * Every message has a reference count that it is initialized to 1
 * at creation. Every Listener that stores a reference to the message
 * increases the reference by one. Every Listeners that does not need
 * the message any more decreases by one the reference. The Channel
 * is responsible for handling the reference counts. Once there are no
 * more Listeners using the message, the reference count is set to zero
 * and the MessageFactory relases the memory.
 * Note that CORBA::release() works in a distinct way. CORBA keeps a
```

```
 * reference count for every object in the server, and a different
 * reference count in every client. The clients increase the reference
 * counts of their proxies, but not of the server. CORBA::release() called
 * in a client does not affect a server.
 * @see DTMS_Source, DTMS_Channel, DTMS_Listener
 */
interface DTMS_Message : DTMS_Object
{
  void increaseReferenceCounter();
  void decreaseReferenceCounter();
  long getReferenceCounter();
};

typedef sequence<DTMS_Message> DTMS_MessageList;

/** The Time Message
 */
interface DTMS_TimeMessage : DTMS_Message
{
  DTMS_Time getValue();
  void setValue( in DTMS_Time time );
};

/** The Sensor reading data
 */
struct DTMS_SensorReadingInfo
{
  long      sensorID  ;
  long      count     ;
  double    occupancy ;
  double    speed     ;
  double    fromTime  ;
```

```
  double   toTime    ;
};

/** Sensor reading message
 */
interface DTMS_SensorReadingMessage : DTMS_Message
{
  DTMS_SensorReadingInfo getValue();
  void setValue( in DTMS_SensorReadingInfo info );
};

/** The Guidance data
 */
struct DTMS_GuidanceInfo
{
  long directionID  ;
  double fromTime  ;
  double toTime    ;
  double travelTime;
};

/** Guidance message
 */
interface DTMS_GuidanceMessage : DTMS_Message
{
  DTMS_GuidanceInfo getValue();
  void setValue( in DTMS_GuidanceInfo info );
};


#endif
```

— End of DTMSMessage.idl —