Final Research Report
Research Project T9903, Task 10
IVHS Backbone Design and Demonstration

ITS BACKBONE DESIGN AND DEMONSTRATION

by

Daniel J. Dailey, Mark P. Haselkorn, Po-Jung Lin
ITS Research Program
College of Engineering, Box 352500
University of Washington
Seattle, Washington 98195-2500

Washington State Transportation Center (TRAC)
University of Washington, Box 354802
University District Building, Suite 535
1107 N.E. 45th Street
Seattle, Washington 98105-4631

Washington State Department of Transportation
Technical Monitor
Pete Briglia
Advanced Technology Branch Manager

Sponsored by

Washington State Transportation Commission
Department of Transportation
Olympia, Washington 98504-7370

Transportation Northwest (TransNow)
University of Washington
135 More Hall, Box 354802
Seattle, Washington 98195

and in cooperation with
U.S. Department of Transportation
Federal Highway Administration

September 1996

TECHNICAL REPORT STANDARD TITLE PAGE

1. REPORT NO.	2. GOVERNMENT ACCESSION NO.	3. RECIPIENT'S CATALOG NO.
WA-RD 411.1 / TNW 96-01		

4. TITLE AND SUBTITLE	5. REPORT DATE
ITS BACKBONE DESIGN AND DEMONSTRATION	September 1996
	6. PERFORMING ORGANIZATION CODE
7. AUTHOR(S)	8. PERFORMING ORGANIZATION REPORT NO.
Daniel J. Dailey, Mark P. Haselkorn, and Po-Jung Lin	
9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. WORK UNIT NO.
Washington State Transportation Center (TRAC) University of Washington, Box 354802 University District Building; 1107 NE 45th Street, Suite 535 Seattle, Washington 98105-4631	11. CONTRACT OR GRANT NO. T9903, Task 10
12. SPONSORING AGENCY NAME AND ADDRESS	13. TYPE OF REPORT AND PERIOD COVERED
Washington State Department of Transportation Transportation Building, MS 7370 Olympia, Washington 98504-7370	Final research report
	14. SPONSORING AGENCY CODE

15. SUPPLEMENTARY NOTES

This study was conducted in cooperation with the U.S. Department of Transportation, Federal Highway Administration.

16. ABSTRACT

Traffic congestion is an increasing problem in many areas of Washington state. Efforts to control traffic flows and mitigate congestion must rely on the ability to accurately monitor the state of traffic flow on highways and arterials. We present a conceptual framework for ITS development to monitor traffic conditions and show how this framework solves numerous high-level problems associated with ITS development. A sample instantiation, the Backbone project, demonstrates the viability of our unified ITS conceptual framework and shows that such a framework can be implemented at reasonable cost and with a high likelihood of successful operation. This project was sponsored by WSDOT and executed at the University of Washington.

17. KEY WORDS	18. DISTRIBUTION STATEMENT
traveler information, backbone, distributed computing, client server	No restrictions. This document is available to the public through the National Technical Information Service, Springfield, VA 22616

19. SECURITY CLASSIF. (of this report)	20. SECURITY CLASSIF. (of this page)	21. NO. OF PAGES	22. PRICE
None	None	50	

DISCLAIMER

The contents of this report reflect the views of the authors, who are responsible for the facts and accuracy of the data presented herein. This document is disseminated through the Transportation Northwest (TransNow) Regional Center under the sponsorship of the U.S. Department of Transportation UTC Grant Program and through the Washington State Department of Transportation. The U.S. government assumes no liability for the contents or use thereof. Sponsorship for the local match portion of this research project was provided by the Washington State Department of Transportation. The contents do not necessarily reflect the views or policies of the U.S. Department of Transportation or Washington State Department of Transportation. This report does not constitute a standard, specification or regulation.

TABLE OF CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

The management of large, complex systems (e.g., power distribution systems, transportation systems) has been greatly aided by the use of increasingly sophisticated sensors and communication technologies to obtain and process real-time data on system activity. Typically, these data are generated by different types of sensors geographically distributed throughout the system. These various sensors take measurements of different system parameters, each of which provides a partial picture of the situation under examination. Under such a system, data typically move from the sensors, through some applications, to the ultimate end user. These data are then used for multiple purposes, including management and control, billing, end-user information, planning, and emergency response.

A wide range of developments have contributed to innovative Intelligent Transportation Systems (ITS). As the ITS community moves toward major demonstration projects and a national architecture, a unifying conceptual framework must be defined. A successful conceptual framework must not only address ITS communication needs, but must also address a number of complex, higher-level needs. These include the following:

(1) the need for inter-agency and multi-jurisdictional data sharing without disruption of existing operations;

(2) the need to support existing investments in ITS technology and system development;

(3) the need to effectively manage the expansion of sensor technologies and user applications in a straightforward, principled manner;

(4) the need to encourage future innovation by providing an open architecture; and

(5) the need for interoperability among local, regional, and national ITS developmental efforts.

We present a conceptual framework for ITS development that has a modular design and addresses the higher-level needs mentioned in the introduction. Again, to meet higher-level needs, the architecture must share multi-jurisdictional data, support existing ITS technology, expand easily, encourage future innovation, and be interoperable among local, regional, and national ITS developments. Our ITS framework has several major components and is based on a client/server distributed-computing model. The components include (1) "instance servers" that bridge data sources to a communication network, (2) "fusion servers" that gather and operate on various data types, and (3) presentation systems for delivering management and traveler information. Each client or server process exists in a distributed-computing environment, so that they might all be present on one computer or distributed over a set of computers. This distributed architecture allows for the system to be "scaled" and also permits any process to migrate to any computer. Similarly, the number and type of data sources or display applications can easily be extended.

The approach described in this paper embodies several specific features, which include (1) an architecture that will deliver real-time data to a large number of consumers while maintaining a mechanism for authorization; (2) a definition of distributed applications that includes an entire set of components arranged in an hierarchical structure; and (3) a clear mechanism for building "value-added" applications that use a generally available data stream. In this paper, we have described in detail a set of paradigms to implement such distributed applications and have discussed several actual implementations in the domain of ITS.

We have presented a conceptual framework for ITS development and have shown how this framework solves numerous high-level problems associated with ITS development. A sample instantiation, the Backbone project, further demonstrates the viability of our unified ITS conceptual framework and shows that such a framework can be implemented at reasonable cost and with a high likelihood of successful operation.

1. INTRODUCTION

The management of large, complex systems (e.g., power distribution systems, transportation systems) has been greatly aided by the use of increasingly sophisticated sensors and communication technologies to obtain and process real-time data on system activity. Typically, these data are generated by different types of sensors geographically distributed throughout the system. These various sensors take measurements of different system parameters, each of which provides a partial picture of the situation under examination. Under such a system, data typically move from the sensors, through some applications, to the ultimate end user. These data are then used for multiple purposes, including management and control, billing, end-user information, planning, and emergency response.

As sensors and the communication infrastructure continue to advance, software developers will more often find themselves working in distributed-computing environments with the potential to access and combine multiple types of real-time data in order to deliver services to clients. These developers need to address a range of issues, including the following:

(1) Sensors vary greatly by the technology employed, their complexity, and location. This implies that a method is needed to make data available on the network in a form that can be consistently interpreted, and this method must accommodate multiple technologies.

(2) Access to data is necessary for numerous, geographically distributed users for multiple purposes. A single data source must be simultaneously accessible to various types of distributed clients.

(3) The number and type of sensors used changes over time, and new sensors will be added. Data structure definitions change with the sensors employed, and the data structures used to interpret the data stream must track these changes.

(4) A single type of data provides only a partial picture of the current state of a large system. Multiple data sources are used with data fusion algorithms to estimate a system state.

(5) Clients require access to data at geographically distributed sites.

(6) Geographically distributed sensors are likely to be located in politically distinct jurisdictions, and data clients are likely to come from politically distinct organizations, necessitating reliable and flexible mechanisms for data security.

(7) Data must be available in real time.

(8) Software supporting these types of distributed systems must be reusable to leverage previous development efforts.

A distributed application can be defined as "an application program consisting of several cooperating components running on different physical nodes." (Schill 1990) In this paper, we describe a structured approach to developing real-time, distributed-computing applications that are suitable for relaying dynamic data in real time to a large but authorized group of users. This approach effectively and economically addresses all the above issues. The approach is based on an asynchronous, distributed, client/server architecture and relies on the creation of autonomous, reusable pieces of software. Within this approach, applications are seen not as clients, but rather as configurations of the various structured software components.

Our structured approach is suitable for creating distributed applications that address a specific class of problem. The types of distributed applications for which our development environment is most appropriate include those that: (1) require multiple, real-time data sources, (2) involve unidirectional data flow, (3) handle data with content and structure that may vary slowly with time (on the order of minutes), (4) require scaling to support thousands of clients, (5) require each client to be identified as an authorized consumer of the data, and (6) require real-time data on a time scale of tenth's of seconds. These features are typically present when data from a variety of remote sensors are

combined to provide surveillance or control functions. This general class of problems is becoming more prevalent with widespread access to the Internet and the development of new sensor technologies.

A wide range of developments have contributed to innovative Intelligent Transportation Systems (ITS). As the ITS community moves toward major demonstration projects and a national architecture, a unifying conceptual framework must be defined. A successful conceptual framework must not only address ITS communication needs, but must also address a number of complex, higher-level needs. These include the following:

(1) the need for inter-agency and multi-jurisdictional data sharing without disruption of existing operations;

(2) the need to support existing investments in ITS technology and system development;

(3) the need to effectively manage the expansion of sensor technologies and user applications in a straightforward, principled manner;

(4) the need to encourage future innovation by providing an open architecture; and

(5) the need for interoperability among local, regional, and national ITS developmental efforts.

Many of the challenges to be addressed by an ITS conceptual framework are, at their root, data exchange problems. Data exchange is central to any ITS application for a number of reasons. First, only one dimension of the overall transportation picture can be presented by a single type of data, whether it is inductance, loop-based volume and occupancy readings, satellite-based GPS tracking, radio-based vehicle location, CCTV-based visual information, laser-based vehicle identification, or data from other real-time sensor technologies. Second, these real-time data sources need to be available in parallel with other less dynamic, but nevertheless essential, data sources, such as map databases, construction site information, and transit schedules. Third, these data systems can fail or

present improbable data, so multiple data sources can help to assure the reliability and accuracy of the overall transportation picture.

Another key aspect of ITS data exchange is that these multiple data sources are often controlled and used internally by politically and geographically distinct agencies and jurisdictions. In addition, the pool of data made available through data exchange must promote the innovative development of future applications, and at the same time must be scaleable for local, regional, and national ITS data needs.

For data exchange to be effective, three general problems must be solved: (1) bringing together the various available data sources on a single communication network accessible by all participating parties; (2) fusing the multiple data types into a reliable, coherent, and dynamic pool of transportation data; and (3) supporting applications that convert this pool of data to information that can be used by traffic engineers and planners, transit agencies, commercial vehicle operators, government officials, and travelers to more efficiently use existing transportation facilities. These problems can all be addressed through a unifying communication framework. In this paper, we explore the state-of-the-art developments in communication frameworks for transportation applications; we describe a conceptual framework for all ITS applications; and we conclude with an example of an ITS project that illustrates this conceptual framework.

2. STATE-OF-THE-ART FOR COMMUNICATION FRAMEWORKS IN TRANSPORTATION APPLICATIONS

This section presents a state-of-the-art review of communication frameworks used in transportation applications as of fall 1993.

Although motorized vehicles are the key components for transportation in North America, research on advanced vehicle-highway systems slowed dramatically in the 1980s, leaving the United States and Canada without national guidelines for a communication framework. As a result, both academia and industry have designed many separate ITS projects, but these modular technologies have not been consolidated in large ITS demonstrations. However, "programs in Japan during the past few years have jolted the United States, leading to a revival of its activities in advanced vehicle-highway technology." (Chen 1990, 695) Several researchers have now developed prototype communication systems for transportation applications.

Although researchers have just begun to develop communication systems for transportation, communication systems are already well-developed in several other applications. ITS researchers can learn from the research in other areas and can use existing communication paradigms in ITS applications. Research for other applications indicates that an ITS architecture must be flexible, open, and, above all, able to support the modular nature of ITS applications (Hsin 1992).

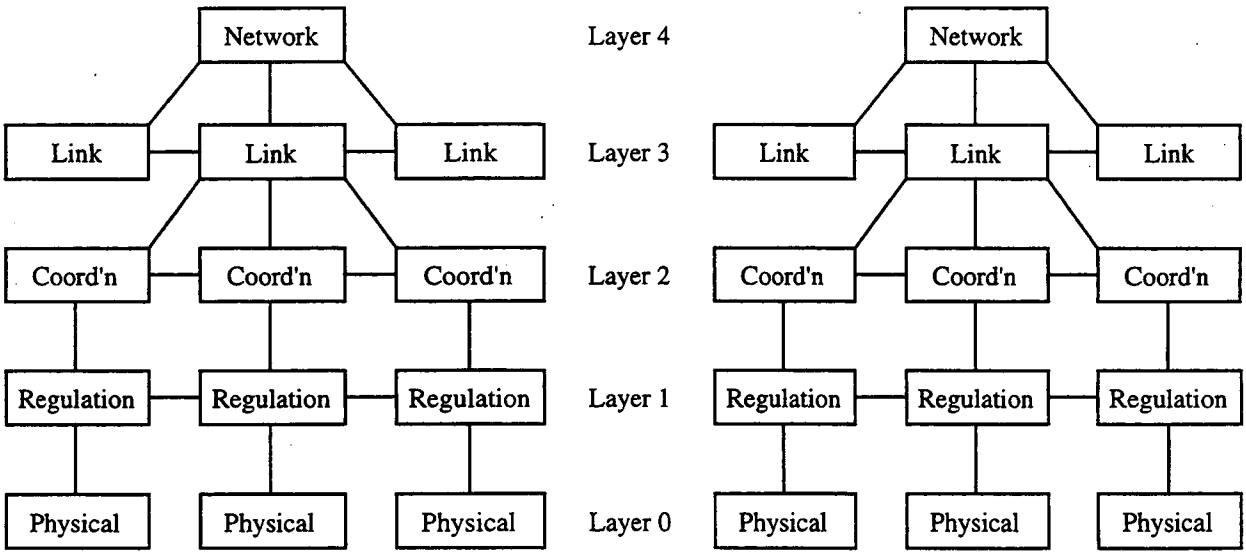
One example of such an architecture is the communication architecture developed by Laraqui et al. for the Integrated Road Transport and Traffic Environment (IRTE), Europe's version of ITS. The IRTE architecture consists of three layers - application, network, and subnetwork - to prevent the system from relying on one monolithic structure. The layers, and the relationships of the applications among the layers, form a "key towards openness (and thus competition) in the provision and combination of IRTE communication facilities and applications services." (Laraqui 1994, 379) Although each layer must operate in

coordination with the other layers to keep information flowing, all layers can perform their tasks independently. The architecture is designed so that each layer "represents one task for transmission and reception. A single task avoids the efforts of solving intralayer communications and reduces the number of tasks in the system." (Laraqui 1994, 382)

Similarly, Varaiya and Shladover have designed an ITS systems architecture that incorporates a hierarchy of five layers. Their hierarchy gives the communication framework a modular approach for the following two reasons: (1) "Each layer presents a standard reference model to the layer above it....and the design of each layer can proceed independently using the reference model of the layer below;" and (2) "communication takes place only between adjacent layers and between peer layers." (Varaiya 1991, 909) Varaiya

Figure 1. Five layer architecture

and Shladover designed their architecture to work for either an Advanced Vehicle Control System (AVCS), in which the vehicle is fully automated, or an Advanced Traffic



Management System (ATMS), in which the vehicle is only partially automated.

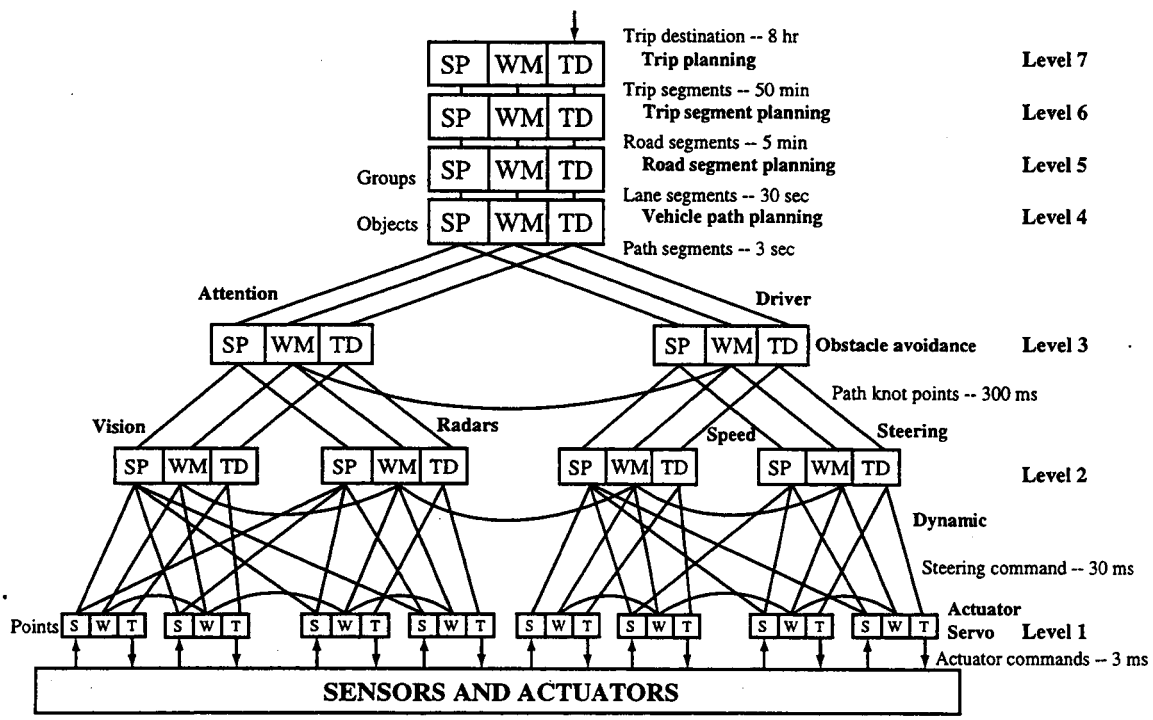
In their architecture, the five layers correspond to five task domains: network, link, coordination, regulation, and physical. The network layer decides the optimum steady state

for traffic flow over a section of the highway. The link layer chooses the lanes that vehicles should drive in to balance the traffic and prevent lanes from being used inefficiently. The coordination layer communicates among the vehicles to properly implement lane changes. The regulation layer maintains proper vehicle orientation and spacing relative to the other vehicles. The physical layer collects the sensor data from the accelerator, brake, and steering wheel. See Figure 1, taken from Varaiya and Shladover's report. (Varaiya 1991, 917)

Still another example of a modular communication architecture in an ITS application is the model designed by Albus, Jeberts, and Szabo. They adapted a Real-Time Control

Figure 2. Seven layer communication architecture

(RCS) architecture that was already being used by other intelligent control systems. This architecture, like the others described here, relies on hierarchical layers of control to achieve



modularity. Its control architecture was designed for an Advanced Vehicle Control System (AVCS) and includes seven layers: trip-destination planning, trip-segment planning, road-

segment planning, vehicle-path planning, steering and attention coordination, steering dynamics, and actuator servos. See Figure 2, taken from Albus, Jeberts, and Szabo's report. (Albus 1992, 383)

In the first three levels, a plan is developed for the level's domain, the plan is decomposed into a series of time segments, and the decomposed goals are passed to the next level. For example,

> input to level 7 is the destination or goal and desired route for the vehicle for a period of up to 8 hours in the future....level 7 generates a plan for the next 8 hours consisting of a series of up to ten trip segments of about 50 minutes in duration....Input to level 6 defines the destination, or goal, that is desired for the vehicle about 50 minutes in the future....level 6 generates a plan for the next 50 minutes consisting of a series of about ten road segments of about 5 minutes in duration....Input to level 5 is the destination, or goal, that is desired for the vehicle about 5 minutes in the future....level 5 generates a plan for the next 5 minutes consisting of a series of about ten land segments of about 30 seconds in duration. (Albus 1992, 379)

The decomposition of high-level goals continues on down through the hierarchy.

> In the Task level (4), individual vehicle goals are decomposed into subgoals for each subsystem. This decomposition process continues through the E-Move level (3) for path planning, the Primitive level (2) for vehicle dynamics, and Servo level (1) for actuator control. As a result of this entire process, drive signals are generated for each actuator that accomplish the highest level goal. (Albus 1992, 379)

In summary, a communication framework is the backbone needed before diverse ITS technologies can work together in one ITS demonstration. ITS researchers are designing prototypes of ITS communication systems that all have one characteristic in common: the frameworks use layers of communication to achieve modularity.

3. A CONCEPTUAL FRAMEWORK FOR ITS SYSTEM DEVELOPMENT

We will now present a conceptual framework for ITS development that has a modular design and addresses the higher-level needs mentioned in the introduction. Again, to meet higher-level needs, the architecture must share multi-jurisdictional data, support existing ITS technology, expand easily, encourage future innovation, and be interoperable among local, regional, and national ITS developments. Our ITS framework has several major components and is based on a client/server distributed-computing model. The components include (1) "instance servers" that bridge data sources to a communication network, (2) "fusion servers" that gather and operate on various data types, and (3) presentation systems for delivering management and traveler information. Each client or server process exists in a distributed-computing environment, so that they might all be present on one computer or distributed over a set of computers. This distributed architecture allows for the system to be "scaled" and also permits any process to migrate to any computer. Similarly, the number and type of data sources or display applications can easily be extended.

The framework is based on a communication backbone - an underlying medium that supports communication between ITS client applications, intermediate processes, and data sources without regard to physical location. Figure 3 represents the central nature of the backbone. This backbone meets both system and higher-level needs. For example, the backbone is an element necessary for distributed-computing applications. But it also promotes inter-agency and inter-jurisdictional cooperation by making it easier for an agency to share its data, and access the data of others, without disrupting that agency's current operations.

We propose that the communication backbone be based on the protocols and standards used by the worldwide Internet community. Our selection of standard Internet protocols not only leverages the vast investment already made by the U.S. government, but

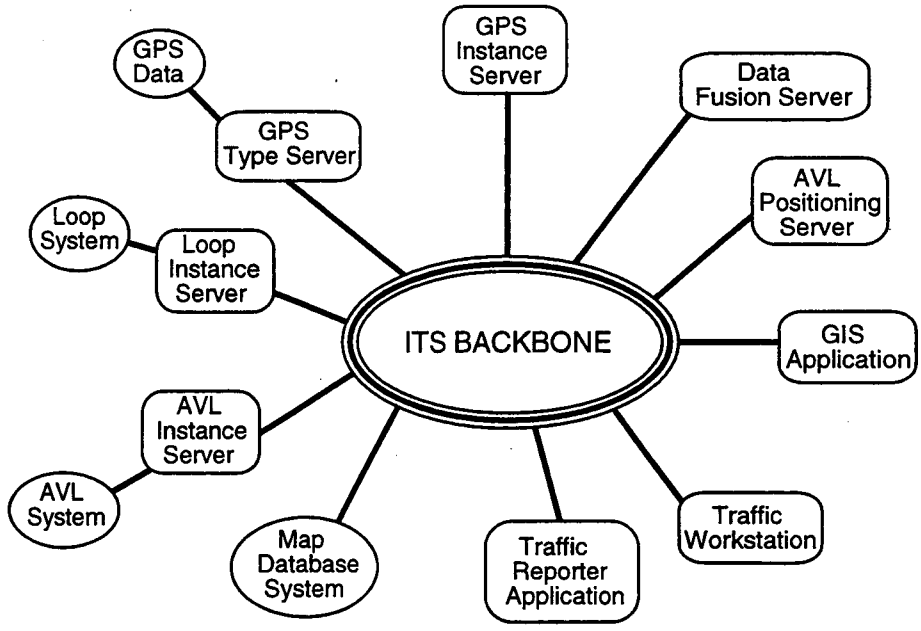


Figure 3. Overall ITS communication architecture

it also enables us to use standard network interface hardware and software. This allows ITS application and sensor developers to focus on ITS issues rather than on communication issues. In addition, this style of network model allows the concepts layered on top of such a backbone to work equally well at the local, regional, and national levels.

A second key aspect of this network model is peer-to-peer communication. This paradigm is often used to share data in a distributed-computing environment, and it inherently supports the notion of geographic independence for ITS data sources and applications. In our framework, each process on the network is capable, with some adjudication, of communication with any other process without regard to its status as a source, intermediary process, or consumer of data. Thus, in Figure 3, each of the client/server processes represented by rectangles can communicate with any other.

This peer-to-peer concept implements a network-wide mechanism for intercommunication. Peers exist independently of agency and jurisdictional boundaries. This allows ITS application developers to access data at any level of complexity, from raw

sensor data to information resulting from data fusion. In addition, applications can access data from any agency or organization willing to share all or some portion of the data they use internally. Because the reward for sharing data is access to everyone else's (as well as to fusion processes and user applications on the network), and because data can be shared without changing the way they are internally used, the principle of "enlightened self-interest" will assure a wide range of data sources and users.

A third key aspect of this network model is peer location using a distributed database of names. In our framework, data sources are advertised on the backbone using a name space paradigm. To gain access to a specific type or instance of data, an application requests the network address of data from a name server. The name server responds with the address of the adjudicator for that service. The adjudicator performs the necessary handshake to establish the peer-to-peer connection between the requester and the requested service. In addition to providing access, this mechanism also provides security because autonomous agencies can filter which data are publicly available by restricting the names supplied to the name server. Even if the name is available, the adjudicator can provide a second level of security by checking an approved access list. In addition to security, this approach also provides a mechanism for data availability on a local, regional, or national scale.

A fourth key aspect of this network model is the use of a client/server concept for inter-application communication. This concept identifies the source and destination of information and data for any specific transaction but does not enforce this definition outside the transaction considered; hence, a particular process can have both client and server attributes. For example, a type server that agglomerates data from probe vehicles is a client of the instance servers that actually obtain the information from the individual vehicles, but it is also a server for downstream applications that use the consolidated probe vehicle data to present a picture of congestion.

The importance of the client/server model is that it provides autonomy for both the data providers and the application developers. An application developer need only know the format of the data expected from the server and some mechanism to find this server on the network. In system terms, this provides a layer at which data structures for inter-process communication are defined. In terms of higher-level goals, new sensors and applications can easily be added. Innovative development is encouraged because developers have a clear model of how to get data and what those data will look like, without having severe limitations placed on what they are allowed to do.

We conclude this general description of our ITS conceptual framework by reviewing some of the higher-level goals that are central to its design. The framework supports existing investments in ITS infrastructure by providing a clear mechanism for integrating past, current, and future developmental efforts. Additional sensors or applications, both existing and yet-to-be-developed, can either implement a network interface internally or use the instance server abstraction to provide a bridge to the ITS backbone.

Our framework also encourages inter-agency and multi-jurisdictional data sharing by providing, in return, the benefits of a global view of traffic and access to numerous other resources and applications. Equally important, a mechanism is provided for agencies to add themselves to the network at minimal cost and with no disruption of internal operations. Thus, the expense of joining this ITS network is slight in comparison with the value received, thereby providing financial incentive for voluntary membership in the ITS environment.

This framework is also compatible with the national trend regarding information flow. That trend is toward a computing environment that is fully interconnected by a national data highway. Our framework recognizes this trend and proposes to explicitly take advantage of existing resources in this area by providing a clear, distributed computing framework for the development of networked ITS applications. In addition, the use of

Internet protocols leverages the huge investment the government has made in this network technology. The viability of using this framework is demonstrated both by the existence of a large community of academics and business people who use Internet resources as an indispensable part of their day-to-day activities and by the instantiation presented in the next section.

4. ITS BACKBONE PROJECT

To apply this idea for a communication framework, we have designed and constructed a Backbone System that takes occupancy and volume data from the Traffic Systems Management Center (TSMC) in the Puget Sound region, combines this information to generate speed estimates, and distributes it across a communication backbone to many different clients. The Backbone project has two goals: (1) to combine

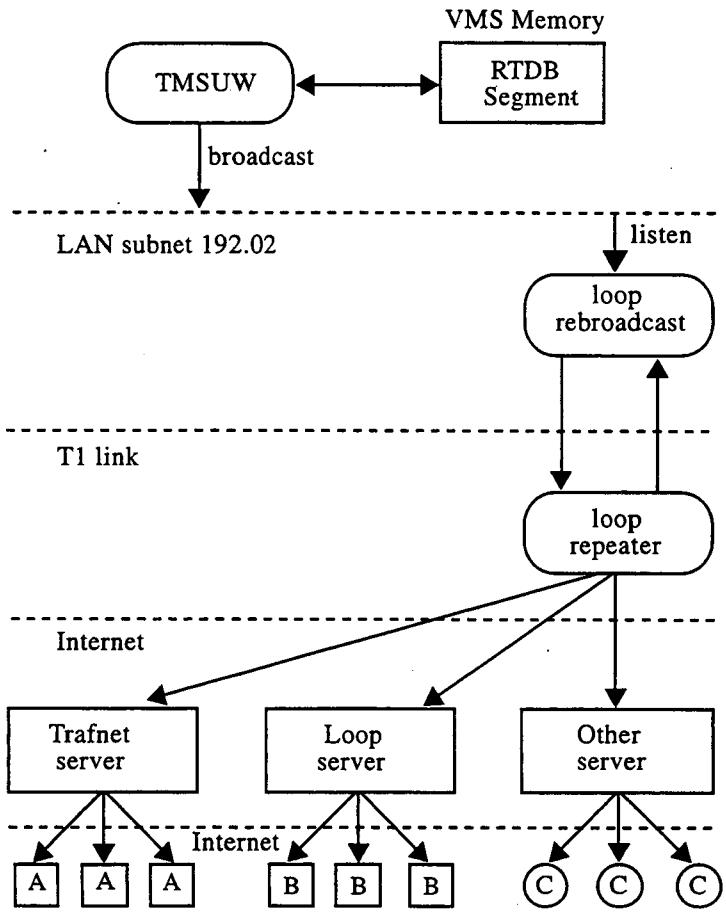


Figure 4. Fusion system architecture

traffic congestion information from all available sources and produce an estimate of traffic speed and (2) to demonstrate the effectiveness of the backbone architecture described in Section 2. Like the other ITS prototype architectures described in Section 1, our framework is implemented in several layers to achieve modularity. The current architecture of the fusion system is illustrated in Figure 4.

There are four major servers - one in each layer - in the system architecture: the TMSUW server, the Loop Rebroadcast server, the Loop Repeater server, and the Loop server. The first server, TMSUW, collects the available loop data and broadcasts those data to the Traffic Systems Management Center's local area network (LAN). Loop Rebroadcast, the second server, collects the data that are broadcast on the LAN and sends the data packet across a T1 link to the Loop Repeater server. The third server, called Loop Repeater, receives data from Loop Rebroadcast and sends the data to the Loop Server. In this action, the Loop Repeater server accomplishes two things: it reduces the load of Loop Rebroadcast, and it allows the transmission between the T1 link to stay within capacity limitations. The last server in the Backbone system is the Loop Server; it transmits occupancy and volume data for each loop and station, and it also transmits the average speed and length for each speed trap. The clients that receive this server's information may actually be other servers that use the data to satisfy the requirements of different end users.

4.1 TMSUW SERVER

The first server, TMSUW, is located on a VMS machine at TSMC called HARLEY. To obtain TSMC's data, the TMSUW server must first understand the structure of the Traffic Management System (TMS) reporting system. The TMS is also running on the VMS machine called HARLEY at TSMC, and upon starting, it builds the following three global databases: a real-time database (TMS_RTDB), a ramp-meter database (TMS_RMD), and a five-minute database (TMS_FMDB). (See Figure 5.) All of these global databases are accessible, but the loop information that is used in the Backbone

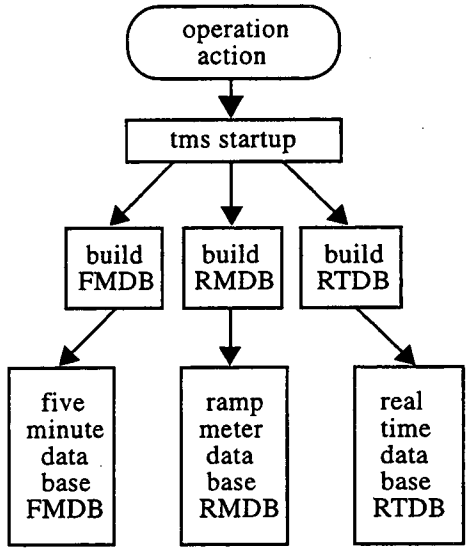


Figure 5. Startup process for TMS

project is obtained only from the Real Time Data Base(RTDB), which is updated every 20 seconds. The other two databases hold the same basic data that are stored in the RTDB, but the data in these databases are normalized for different purposes.

The RTDB contains two parts: a name table and data records. The name table consists of name, type, length, and offset fields. The name field contains all of the loop names currently available in the RTDB; each name is a combination of the specific loop name and the name of the cabinet to which the loop belongs. For example, the name, ES090D:_MN___1,O indicates that the loop is located in cabinet OES090DO and is on a mainline (M) road going north bound (N) in lane number one (1). The second field in the name table, the type field, distinguishes between the three types of loops that collect data: loop, station, and speed trap. Because the different loop types are different sizes, the length field is needed to specify the length of each loop. Finally, the offset field points to the position of the loop data in relation to the other loops in each 20-second data record.

The second part of the RTDB consists of 181 separate data records. Each record holds the data collected by all of the loops during a 20-second period. With 181 individual

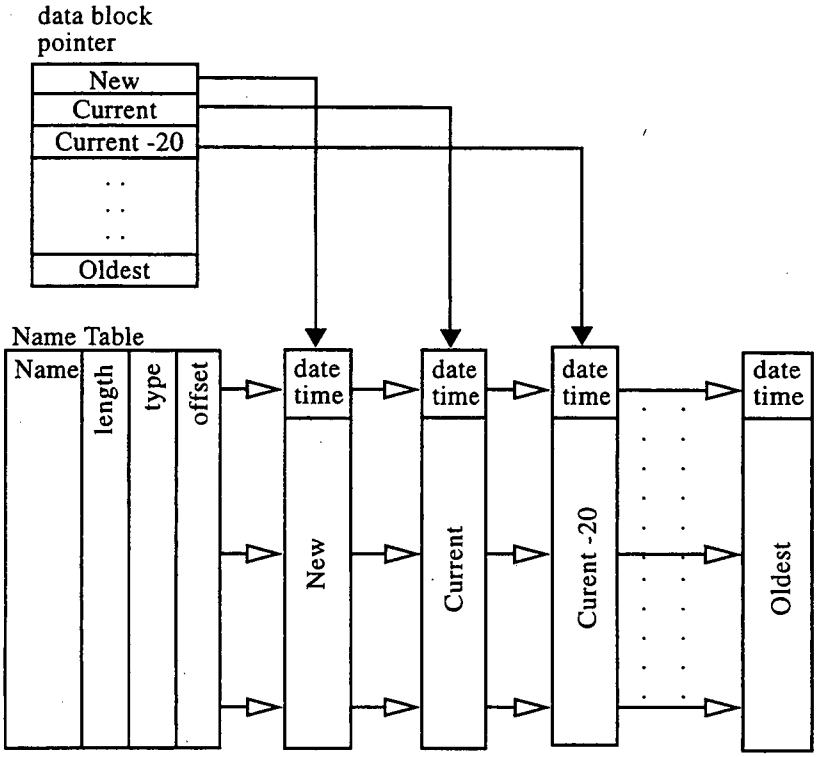


Figure 6. Data structure of RTDB

20-second data records, the RTDB holds one hour's worth of data (181 = 60 min. x 3 blocks of twenty sec. +1). To keep the RTDB updated with data from the most recent hour, the oldest data record is discarded when a new data record is obtained. Specifically, the data record just received from the traffic reporting system is put in the new data block, and all of the other data records shift one slot over, indicating that each data record is an additional 20 seconds older than the newest data. When a data record becomes the oldest record in the RTDB and a new data record is received, the oldest data record is discarded; it is essentially rotated out of the database. To indicate that a new data record has been stored, the RTDB increases the rotation scroll number in global memory. See Figure 6 for an illustration of the two parts, the name table and data records, of the Real Time DataBase.

With knowledge of the RTDB's structure, the TMSUW server uses the name table to access data records. To begin, the TMSUW server reads the name table from the RTDB

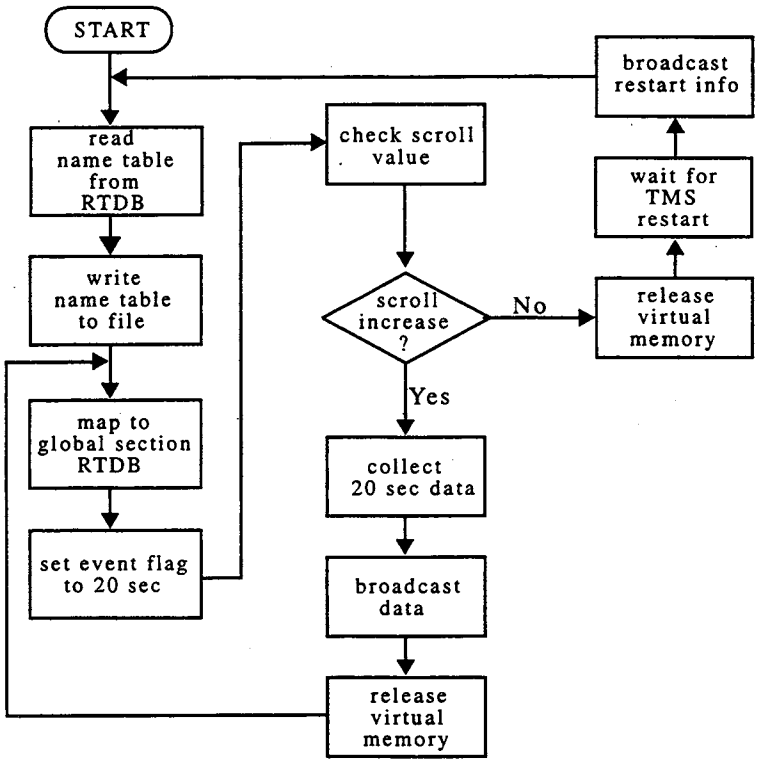


Figure 7. Flow chart of TMSUW

global memory and writes it to a file. Then the server starts the data collection cycle. The cycle begins when the server maps to the global section of the RTDB and sets up the corresponding pointers for each block holding a data record. At the start of the cycle, the server sets the event flag to 20 seconds, the duration of each collection period. In that 20-second cycle, the server checks the global scroll value in the RTDB to determine whether the data records have rotated. If a data record rotation has occurred, it indicates that a new data record has been received and is in the new data block. If a data record rotation has not occurred, the TMS has been reset.

In the first case, when a new data record has been put into the new data block, the server collects the new data record, broadcasts it on the LAN in TSMC, and finishes the collection cycle. At the end of that 20-second cycle, it begins a new collection cycle by checking the global scroll value in the RTDB. In the second case, when the server

discovers that the system has been restarted, it waits a few minutes to ensure that the TMS successfully restarts and then broadcasts a special packet to the LAN. This packet tells the Loop Rebroadcast server that the TMS has been restarted and the file that holds the name table must be updated. Figure 7 shows the flow chart of the process managed by the TMSUW server.

4.2 LOOP REBROADCAST SERVER

The Loop Rebroadcast server resides on a machine called Loops, which is hooked into the TSMC LAN (subnet 192.0.2). The server was placed on the Loops computer rather than TSMC's HARLEY computer to avoid disturbing TSMC's system and slowing down its process. This server listens to the LAN to determine whether a data packet has been broadcast by the TMSUW server, captures any TMSUW data packets, and then sends those data packets across a T1 link to the Loop Repeater server, which is geographically distant from TSMC. The system architecture of the Loop Rebroadcast server is shown in Figure 8.

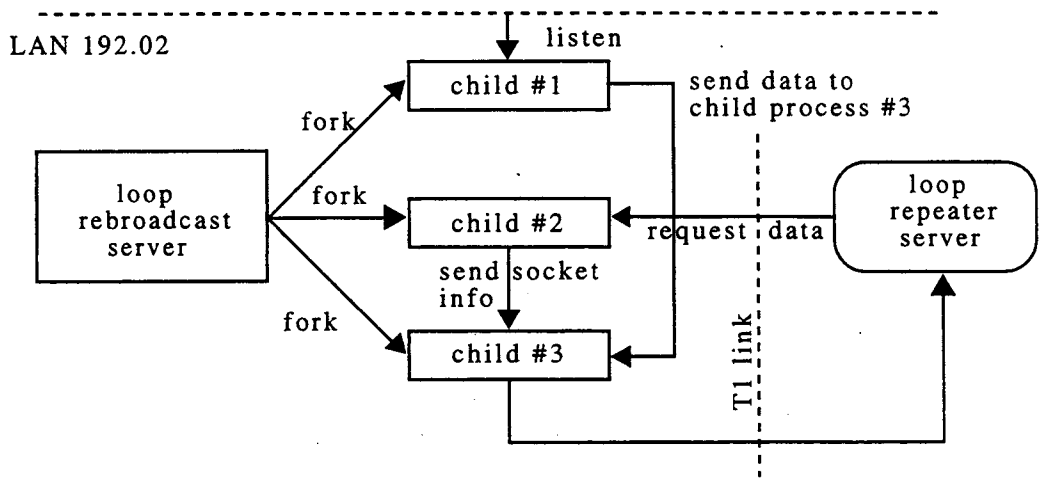


Figure 8. Loop Rebroadcast server system architecture

To accomplish the retrieval and transmission of the data, the loop rebroadcast server initiates three different "child" processes: the first child process monitors the TSMC LAN and retrieves data sent by the TMSUW server; the second child process establishes the connection with the Loop Repeater server; the third child process transmits data to the Loop Repeater server. Of course, before the third child process can send data, it must obtain information that the first and second child processes have gathered, i.e., the loop data and the connection information. Communication among the child processes occurs through a Unix internal socket pipe.

Although the Loop Rebroadcast server sends data to only the Loop Repeater server, it is capable of accepting other connection requests. To handle additional requests, the Loop Rebroadcast server must initiate all three child processes, avoiding conflicts between requests for data and data transmission. For example, while a process is sending the requested data to the client, another client may request a connection at the same time. This second client will not get any response until the process has finished sending the first data request. If the data transmission is long, the second data request could be lost. Therefore, three child processes are required: one to capture loop data, one to listen for requests from clients, and one to transmit the data to the clients.

4.3 LOOP REPEATER SERVER

The Loop Repeater server, which runs on a machine located at the University of Washington, performs a role in the architecture of the Backbone project similar to that played by the Loop Rebroadcast server; they both receive loop data and pass it on to other servers. However, the Loop Repeater server sends the loop data to other servers across the Internet rather than across a T1 link, and this Internet connection gives the server greater capacity for handling a large number of clients. The Loop Repeater server is included in the Backbone architecture primarily to provide this increased capacity. As an added

benefit, several loop repeater servers can be cascaded together to handle hundreds of client requests for data. Figure 9 shows the system architecture of the Loop Repeater server.

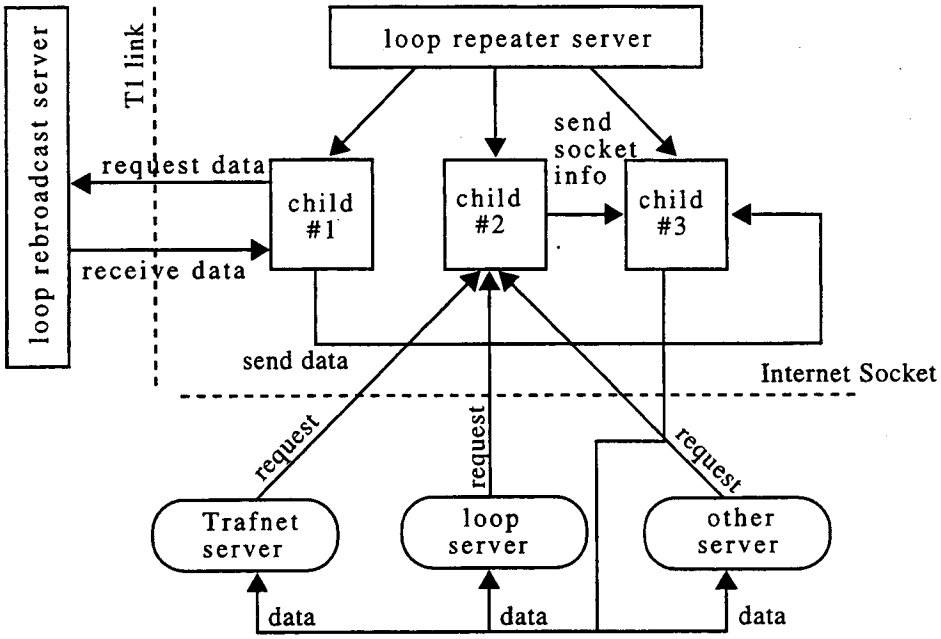


Figure 9. Loop Repeater server system architecture

Just like the Loop Rebroadcast server, the Loop Repeater server initiates three child processes: one to receive loop data, one to listen for requests from clients, and one to transmit the data to the clients. In the Loop Repeater server, child process 1 does not have to monitor a network to capture the loop data. Instead, child process 1 sends the Loop Rebroadcast server a request for data, knowing that the Loop Rebroadcast server will establish a connection and send the loop data to child process 1 as soon as the data have been captured from the TSMC LAN. Child process 2 receives requests for the loop data from other servers, including the TrafNet server and the Loop server. As the last step, child process 3 sends the loop data to all servers with which child process 2 has established a connection.

4.4 LOOP SERVER

The Loop server is responsible for receiving data from the Loop Repeater server, fusing the data, and sending the fused data to other interested servers. To receive and send data, the Loop server uses the same architecture of child processes that is used by the Loop Rebroadcast and the Loop Repeater servers. The system architecture of the Loop server is shown in Figure 10.

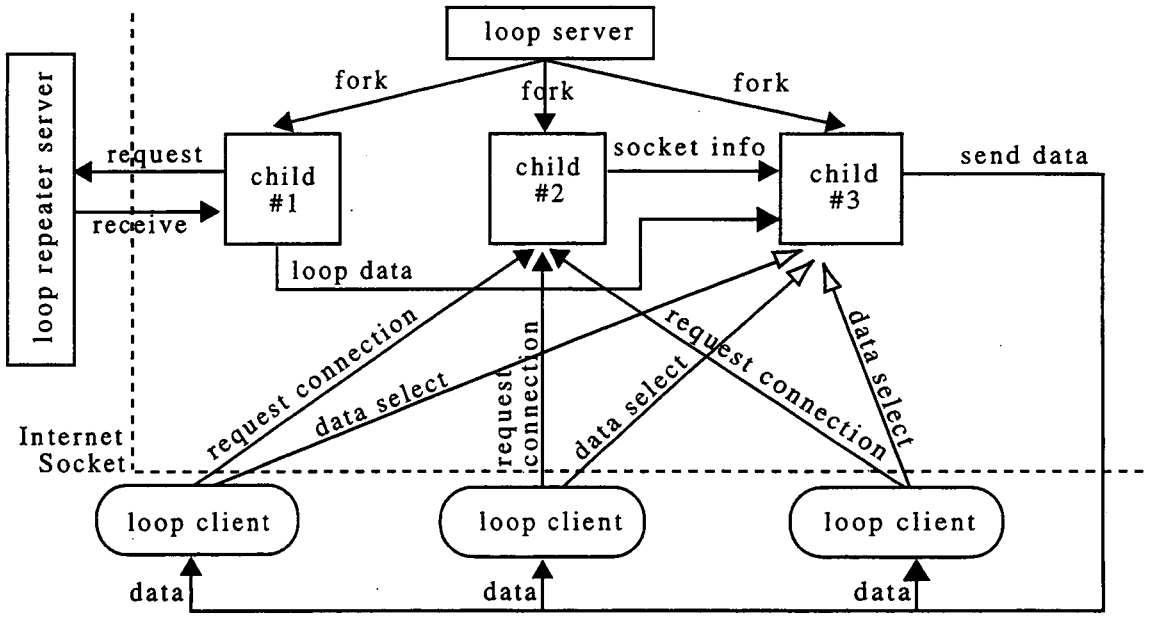


Figure 10. Loop server system architecture

The three child processes in the Loop server architecture function in basically the same manner as the child processes in the Loop Rebroadcast and the Loop Repeater server architectures. The first child process requests the raw data packet from the Loop Repeater server. Upon receiving the data, it sends the data packet to child process 3 via Unix socket pipes.

The second child process of the Loop server receives all the connection requests from end users who are interested in the loop data. After granting a connection to an end

user, it sends the clients' connection information to child process 3 via Unix socket pipes and waits for additional data requests.

Child process 3 receives loop data from process 1 and client connection information from process 2. In addition, the third child process receives requests for specific data directly from the clients. Clients can ask the process to send them only the list of particular loops on a specific route, such as all the available loops on Interstate 5. Acting on all the information that it has gathered, child process 3 sends the requested loop data to each of the connected clients through an Internet TCP socket.

5. GENERAL FRAMEWORK

In the following section, we first present our general framework and then demonstrate our approach by describing several Intelligent Transportation System (ITS) applications we have created and deployed. (It is noteworthy that our paradigms fit within the context of the FHWA ITS Architecture.) (Rockwell, *Theory* 1995; Rockwell, *Physical* 1995)

5.1 PREVIOUS WORK

Previous work has shown that structuring mechanisms aid in managing the complexity of distributed applications. These mechanisms support a number of desirable functions, including transparent allocation of processes to CPUs, support for dynamic changes in the structure of a distributed application, and support for development in the form of tools for common management functions.

The approach presented here provides basic support for the development of a general class of distributed-computing applications that use only widely available system support. Previous work has focused on designing new system support for distributed applications; however, this new support has required the implementation of a significant amount of new, operating system-specific code to make the proposed environments available on every platform. We compare and contrast our approach to three previous efforts: (1) the POLYLITH Software Bus, (2) the Regis environment, and (3) the ConicDraw system.

The POLYLITH Software Bus (Purtilo 1994) is a system developed to allow modules implemented in different languages to run on heterogeneous architectures, interconnected by varying communication media. POLYLITH was introduced in 1985.

The goals of POLYLITH are (1) to separate the specification of an application's structure (the modules present) from the implementation of its component modules, (2) to separate the application's geometry (where modules run) from the implementation of

modules, and (3) to allow the specification of how modules communicate without modification of the modules at the source level.

The first POLYLITH goal is met by providing a Module Interconnection Language (MIL) for specifying the structure of an application. Modules may be implemented in the designer's language of choice. The MIL also allows control of the application geometry, meeting the second goal. Default guidelines allow the designer to omit explicit specification of the geometry. The third goal is met by a mechanism called a *software bus*, an agent that encapsulates run-time interfacing concerns for an application. Changes in the bus are all that are required to change the inter-process communication (IPC) method.

The approach presented in this report meets the first two goals of the POLYLITH system in different ways and with far lower system support requirements. Our approach creates distributed applications from component parts. The components consist of executable images and configuration data, and each component acts as an agent for the execution of other components on a host. The application geometry is contained in the configuration information. We depend on only widely available inter-process communication (IPC) paradigms to build our components and, thus, the components do not readily change the IPC methods.

The Regis programming environment (Magee 1994) is a successor to its developers' earlier CONIC (Magee 1989) and REX (Kramer 1992) systems. It provides a configuration language known as *Darwin* that allows specification of the structure of a distributed application and allows for dynamic instantiation of computational components. It restricts components to implementation as C++ objects, something its predecessor REX did not require. CONIC required that computational components be implemented in Pascal.

In comparison, our configuration language is represented in tabular form, and the tables consist of tuples of component name, site for execution, and configuration information to define connectivity. These tuples are dynamically created and used to

initialize a new application. Our applications are dynamically created but, once instantiated, operate in a static configuration. However, any single component may be dynamically assigned to a portion of several application programs at the time the application is configured.

ConicDraw (Kramer 1989) is a graphical front-end to the CONIC system that allows specification and modification of distributed application geometry at run-time. ConicDraw provides a graphical interface that allows construction of distributed applications from pre-defined module types. Support for diagram layout is provided (rudimentary planar graph drawing).

We specify our configuration geometry using a simple planar graphing application written in tcl/tk to generate the configuration tuples. Icons representing individual components are selected and placed on a pallet, and the connectivity between components is defined by drawing directed lines between them. The configuration is specific before initialization and remains static during the execution of an application. The same configuration tool is used to provide monitoring and status information on individual components and elements of the components in real time. The tool can also be used to halt or restart individual components as deemed necessary by the application architect. A more detailed description of our structured environment follows.

6. THE DEVELOPMENT ENVIRONMENT

In this section, we describe a development environment in which (1) real-time data are available asynchronously from various geographically distributed and technologically independent sources; (2) real-time data are provided to various geographically separated clients on an as-needed basis; (3) real-time data fusion is used to produce additional "value-added" data streams; (4) clients are added through an efficient scaling mechanism that maintains desired security; and (5) the data dictionary of the real-time data changes slowly over time, and these changes are handled in a principled manner.

Our development environment consists of four components, and we define an application as an appropriate configuration of the components. The components constitute a toolkit that implicitly handles messages and communication, freeing developers to deal with the flow and interpretation of data streams.

6.1 COMPONENTS

In this section, we provide an overview of the functions of each component type. Section 6.2 then describes how components are linked to form applications.

Our development environment is composed of software components that can be configured and linked to construct applications. Each component is (1) autonomous, i.e., components are distributed across computing platforms; (2) independent, i.e., components can operate in parallel; (3) a peer of any other component, i.e., components can be placed into an evolving hierarchical matrix with great flexibility; and (4) reusable, i.e., coding is efficient, and much of the application design is reduced to configuring available components.

There are four types of components, each of which affects the flow of data. (See Figure 11.) The component types are (1) a Source Component, which makes a data stream available; (2) a Redistributor Component, which obtains a data stream from one component and redistributes it to one or more other components; (3) an Operator Component, which

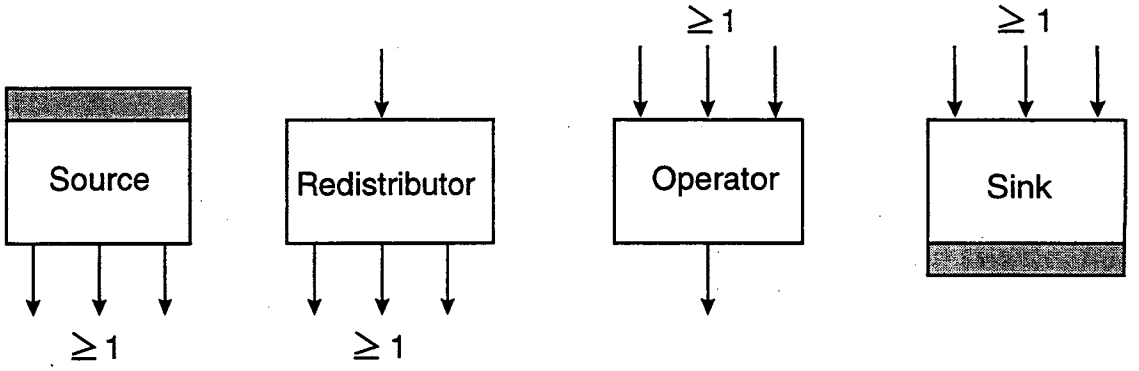


Figure 11. Components

obtains data streams from one or more components and creates a new data stream for distribution to one or more components; and (4) a Sink Component, which obtains data streams from one or more components. Redistributors and Operators function both as client and server, whereas Sources are servers, and Sinks are clients.

We use the existing network inter-process communication (IPC) facilities as a base for implementing the components that make up the building blocks of our application development environment. Unlike previous work that required extensive software for each participating operating environment, we require only a socket interface to an IPC library to implement applications on a wide variety of operating systems at low cost. Our approach uses three IPC mechanisms: (1) the ability to connect to a specific service at a specific location on the network, (2) the ability to answer requests for service from remote clients, and (3) the ability to maintain connections between the components of our applications for data transfer.

6.1.1 Elements of a Component

Before describing these four components in greater detail, we will first describe three elements that are key, reusable pieces of these components. Three of our four components (Source, Redistributor, and Operator) are built from these three specific

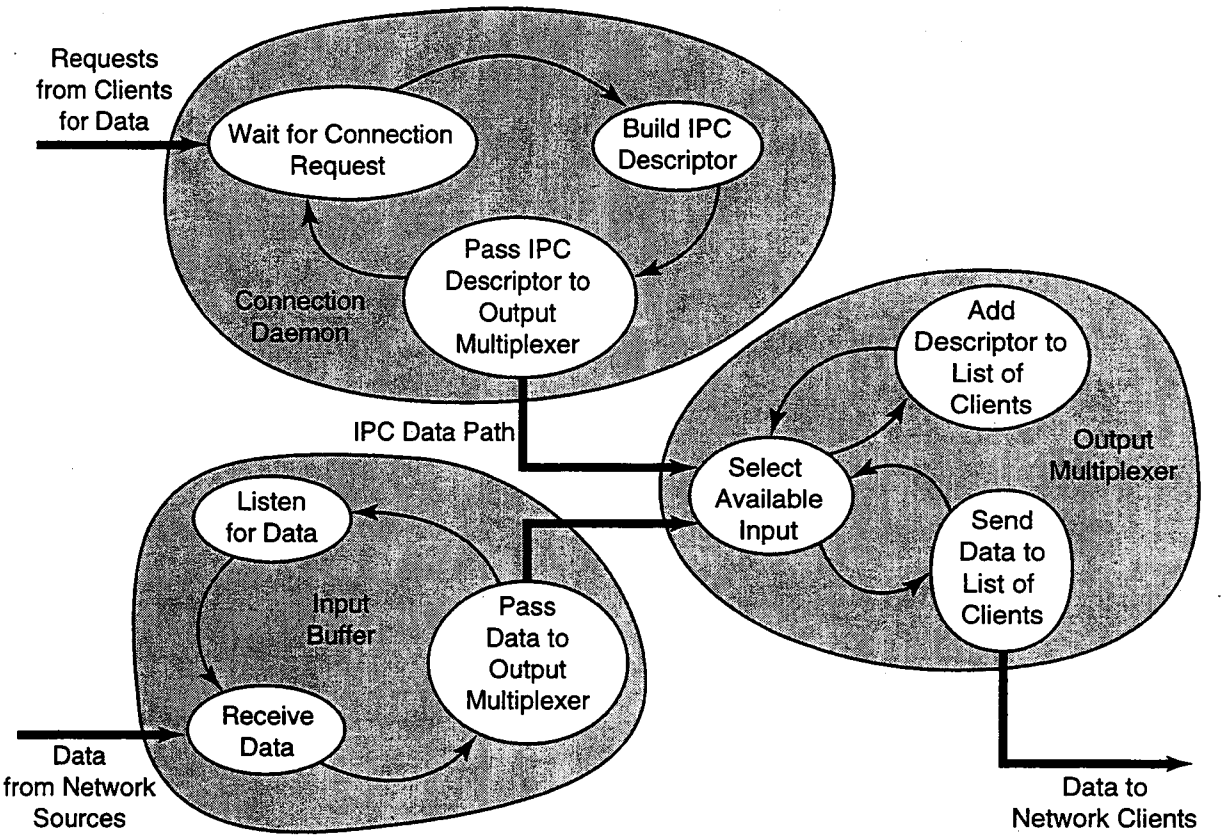


Figure 12. State diagrams for the elements in a redistributor

reusable elements. The elements, and a state diagram for each element, are shown in Figure 12.

The "Connection Daemon" element waits for and accepts asynchronous IPC requests across a network. These requests are for either data or administrative functions. It services these requests by first establishing the authorization of the requester and then either accepting a connection from a client requesting data or executing an administrative function, such as "restart." Once network connectivity and authentication have been established, the information about the IPC connection to the client is passed to the "Output Multiplexer."

The "Output Multiplexer" element maintains a list of connected clients and multiplexes the data stream to all active clients. It also maintains state information in the

form of a data dictionary structure, which is the first information passed to a newly connected client. This data dictionary definition allows the client to interpret the data stream. The data stream to be multiplexed is received asynchronously from the element labeled "Input Buffer."

The Input Buffer initiates and maintains connectivity with a data source and also receives available data from a data source. This element provides a data buffer for the data from a source and forwards the data to the Output Multiplexer. This process guarantees that the component remains synchronized with the data stream from a data source. This synchronization is necessary because the data stream from sensors is sent in a frame and each frame has a fixed length (defined in the beginning of the frame). If synchronization is lost, this element terminates the connection to the data sources and then reestablishes the connection to re-synchronize the data stream. Multiple input buffers may exist in a single component to receive data from several sources.

The Connection Daemon, Input Buffer, and Output Multiplexer all exist on one computer, communicate with each other asynchronously, and operate in parallel. This parallelism is obtained by using operating system support for multiple processes that may time slice one CPU or execute each process on parallel CPUs. The parallel operation of these elements is necessary because of the asynchronous nature of the tasks performed by each element. The elements - Connection Daemon, Input Buffer, and Output Multiplexer - make up the heart of the reusable code in our approach, and three of the four components (the Operator, Redistributor, and Source) use these three elements as building blocks. A more detailed description of each component type and its construction follows.

6.1.2 Source

A Source component exists for each sensor technology and makes the data stream from that technology available on the network. It does this by (1) interfacing with the sensor technology, (2) using knowledge of the content of the data stream to build a self-describing data dictionary into that data stream, and (3) making the data stream available to

other components on the network. Note that the Source component does not include the technology that generates the data stream but, rather, is a tool that allows developers to ignore the nature of that technology. Sensor technologies range from a single sensor with a single, simple data type to computer systems providing rich streams of complex data. Whatever the nature and complexity of the data-generating technology, the related Source component packages its stream of data to give it a peer status with other data streams on the network.

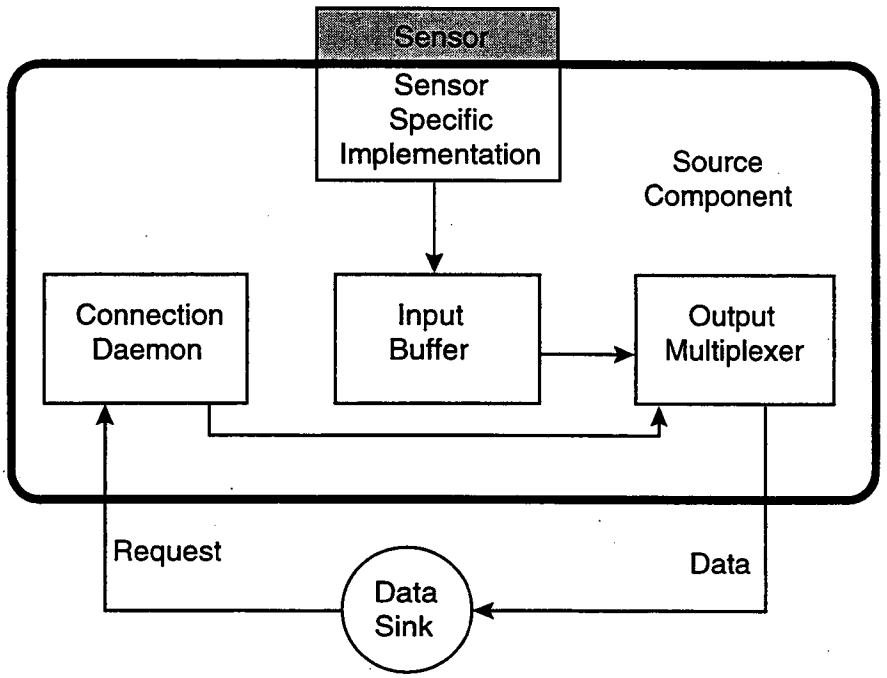


Figure 13. Elements of source component

The Source component type adds one element to the three basic elements just described, namely a process specific to the sensor/system that is the originator of the data stream. (See Figure 13.) This additional process communicates directly with the sensor/system to obtain data and information about the format, contents, and meaning of the data stream. It builds data structures - to define format, contents, and meaning - that are provided to clients as an initial data dictionary before it transmits the sensor data. These

constructs produce a self-defining data stream with an initial dictionary and following data. As the information coming from the sensor changes in format, content, and structure, this component builds a new data dictionary and passes it on to "downstream" components.

6.1.3 Redistributor

A Redistributor takes data from one component and multiplexes it to other components. Each Redistributor is client to a particular data-providing component and allows the data stream from that component to "fan out" to each of its own client components. The Redistributor component has two basic functions: (1) scaling and (2) authorization. In our structured development environment, scaling is achieved by linking redistributors in a hierarchical fashion. Because each Redistributor services *N* clients, *M* layers of Redistributors service *N* x *M* clients. Because components are autonomous, Redistributors can reside on any computer and operate in parallel. In this way, scaling can be economically achieved in fixed-size blocks through the addition of Redistributors operating on parallel computing platforms. (See Figure 14.)

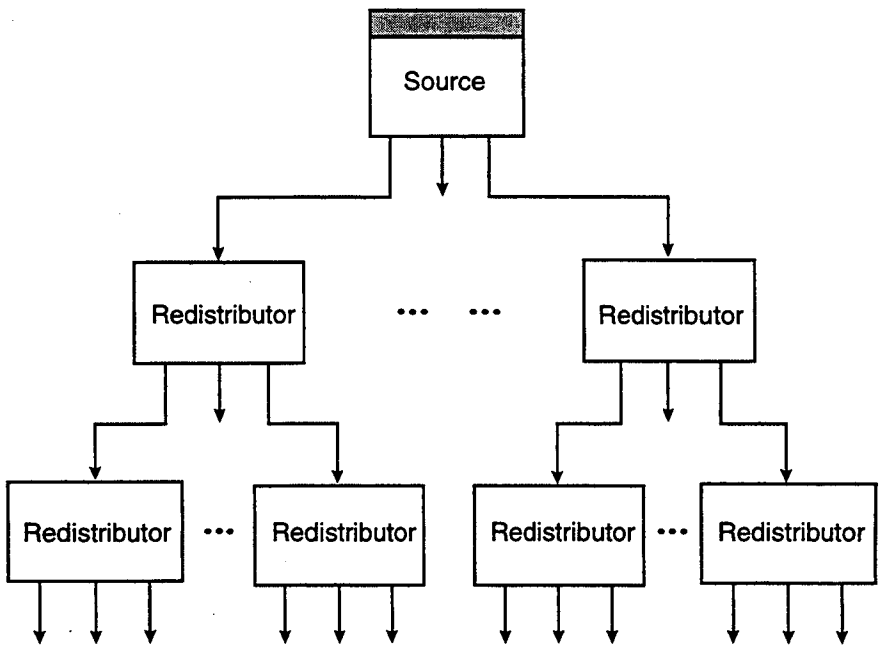


Figure 14. Hierarchical scaling mechanism

The Redistributor also authorizes data access. The Redistributor maintains a dynamic client list and, whenever they are available, passes new data on to clients on that list. When a request for data comes from a component not currently on the client list, the client is checked against an authorization list. This authorization list is part of the initialization information for each Redistributor and may remain as simple as a static list or may include dynamic queries to a remote database. The dynamic or static authorization list and authorization mechanism are part of the configuration information. Similar authorization is performed by Operator and Source components as well.

The Redistributor component type is constructed from the three elements described above, as well as from configuration information, including the data source network location and authorization information listing allowable clients. Figure 15 and a table of configuration information represent the Redistributor component type.

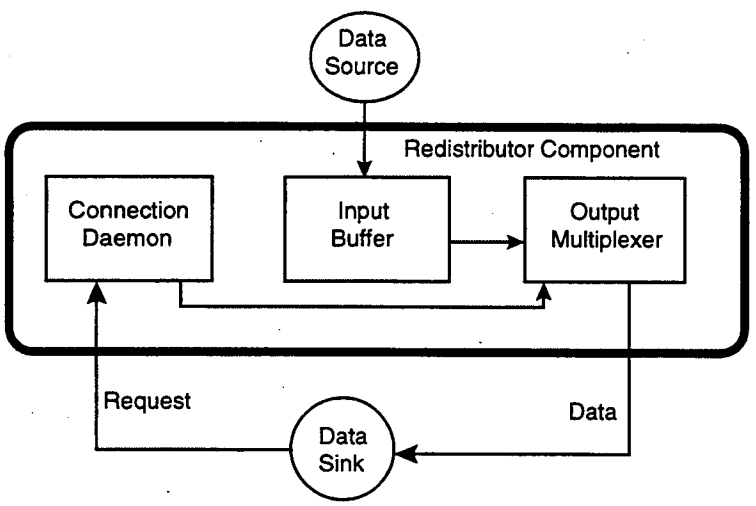


Figure 15. Redistributor component elements

6.1.4 Operator

The Operator component type comprises the three basic elements plus a data fusion activity that operates on the data content as the data are passed from the input buffer to the Output Multiplexor. (See Figure 16.) An Operator receives data streams from one or more components, performs functions on those data streams, and makes available a single,

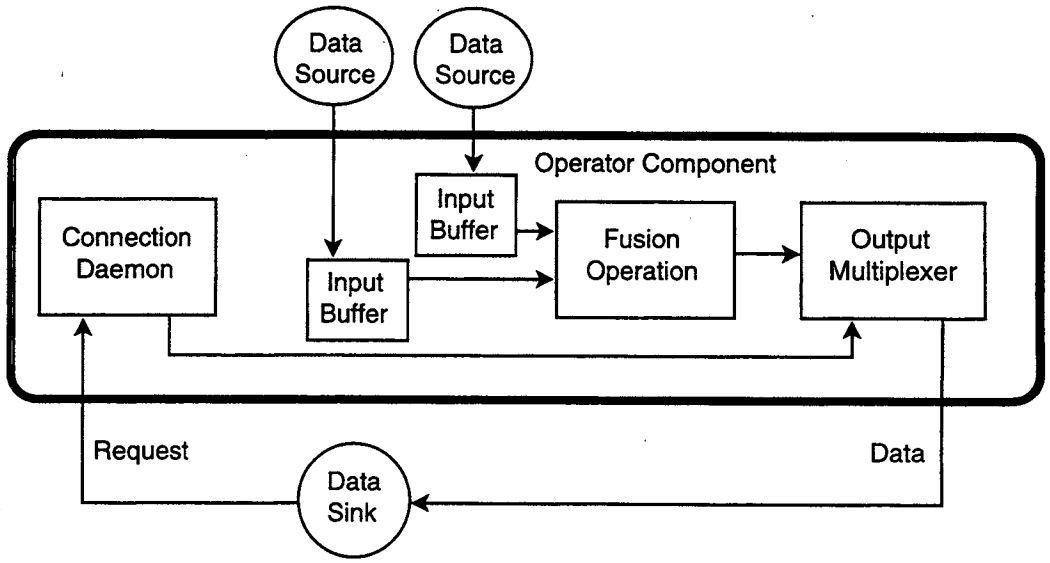


Figure 16. Elements of the operator component

newly created data stream to one or more components. Its general purpose is "data fusion," converting data from the form provided by the serving components to a form desired by the eventual clients.

Each Operator component is associated with a specific data fusion activity. These activities can range from combining like types of data to creating new information from a variety of data sources. Like the Redistributor, the Operator also performs authorizes data access.

6.1.5 Sink

The Sink component is the portion of our approach that is commonly viewed by a user as an application. It is the component that typically either has a user interface or

performs an activity that affects the environment. It usually obtains data from one or more components (such as an Operator, or a Redistributor that is downstream from an Operator). Examples of several types of Sinks are provided in Section 6.4.

6.1.6 Summary

These four component types: (1) Source, (2) Sink, (3) Redistributor, and (4) Operator, are the basis for creating our distributed applications as described in the next section.

6.2 APPLICATIONS FROM COMPONENTS

In our approach, applications are constructed from components by connecting Sources, Redistributors, Operators, and Sinks in a hierarchical structure. An application is composed of all the structures necessary to take original sources of data, fuse the data, and distribute the resulting information to the data sinks. These structures include the individual component types, structural information about the path for data flow through the application, and information necessary to allow the Operators and the Sinks to use the data.

For example, the simplest application is a single-client, single-server model in which a data sink operates on the original source data (e.g., the data sink displays the sensor status or output). This application is composed of two components, a Source and a Sink. The elements of these two components use (1) connection information that allows the sink to connect to the source, (2) authorization information that allows the source to send the data to the sink, and (3) a shared definition of the interpretation of the data stream. Such an application is shown in the shaded area labeled (1) in Figure 17 and is similar to the single-client, single-server notion of distributed applications. In our approach, however, most applications are composed of numerous components.

A common application scenario would include (1) multiple, data-specific sources that are geographically and technically disparate, (2) a Redistributor for each source, (3) an Operator for each type of data, (4) an Operator for each type of information that will be

created from available data products, (5) a Redistributor for each block of sinks, and (6) multiple, special purpose Sinks that require access to data streams to achieve their purpose.

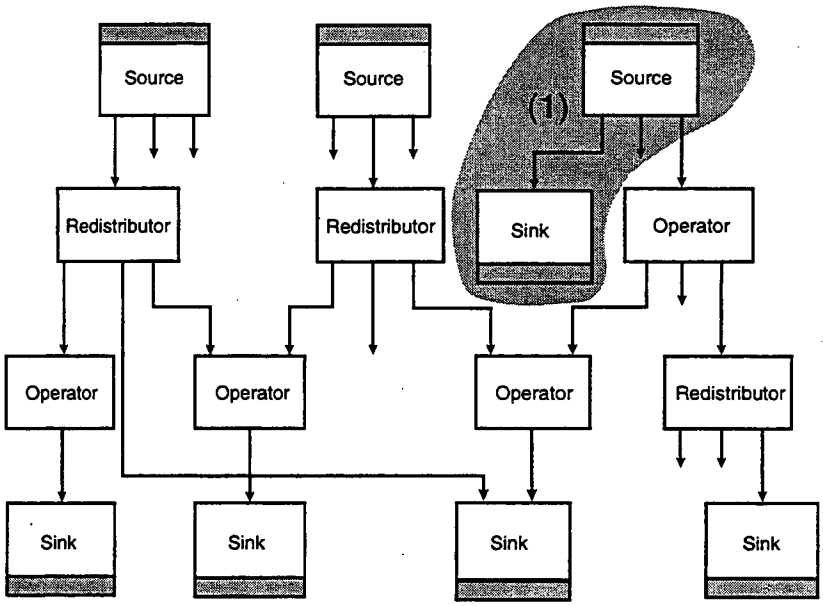


Figure 17. Construction of applications from components

Applications are composed of several of our base components and are initiated with remote procedure calls that start individual components on local or remote computers and initialize these components with their data sources. Section 6.4 presents some examples of applications we have built using the paradigm presented here.

6.3 TOOLS FOR APPLICATION CREATION AND MONITORING

We have created a set of tools to be used within our framework to develop and manage applications. The applications are developed and managed with a set of graphical tools to assemble the hierarchy of components that make up an application, and the same type of graphical tool is used to identify the status of elements within operating components.

To support the implementation of management tools, each of the elements within a component has monitoring facilities that report status to a "simple control and monitoring daemon" (SCMPD) over a connectionless channel. This status consists of static data such as (1) the Source and Sink ports and (2) the address of the server from which it receives data. It also consists of dynamic data such as (1) the time of the last message processed by the component, (2) the status of the component (such as a heartbeat indicator), (3) the present state of the component (initializing, running, shutting down), and (4) the time of the most recent change of state. The SCMP daemon logs this information and also supports connections by a console application.

The console application creates a connection to an SCMP daemon and produces two windows. The first is an acyclic planar graph representing the connections between the components that makes up the applications that are reporting to this SCMP daemon. The second window provides a detailed account of the status of the elements within the components reporting to that SCMP daemon. (Examples of these windows are shown in Figure 18.)

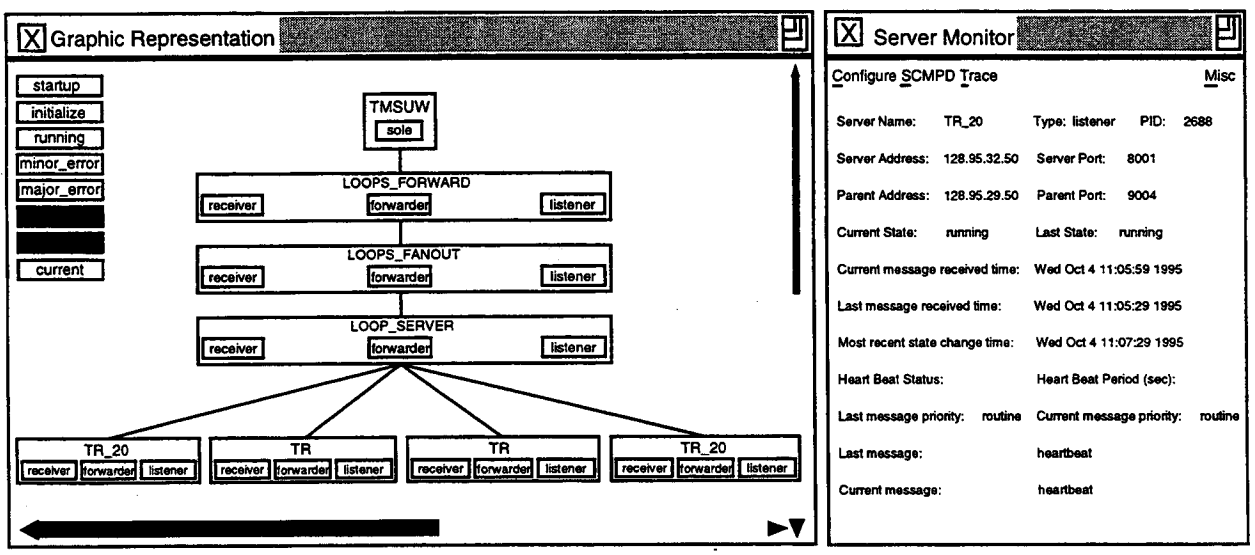


Figure 18. Application console

An application similar to the console is used to create the control tables that instantiate each of the components and to create the connections between the components. An application is created by starting a set of components whose intercomponent communication is defined in terms of the Source and Sink address and port numbers. Examples of ITS applications created using this approach are presented in the next section.

6.4 ILLUSTRATION USING ITS DATA

To illustrate the implementation of our approach, we describe several Intelligent Transportation System (ITS) applications we have constructed using our paradigm. ITS applications are good examples of the type of remote sensor problems for which our approach is appropriate. In ITS applications, there are typically large numbers of sensors that are geographically diverse and controlled by autonomous political and fiscal entities. However, the applications need access to data from a variety of sensors widely deployed and owned by other entities. These sensors typically have update rates on the order of milliseconds to seconds. Two examples of ITS data sources are congestion monitoring systems and probe vehicles.

Congestion monitoring is implemented with a variety of sensor types. In our demonstration applications, we use two types of sensors, *inductance loop sensors* and CCTV cameras. The inductance loop sensors are thousands of wire loops buried in freeways and arterials, including their associated electronics, that can identify properties of vehicles passing over the loops. These properties include the passage of a vehicle, the duration a vehicle is present, count-per-unit time, the average speed of a vehicle, the length of a vehicle, and the type of vehicle. The properties measured vary depending on the available electronics. The CCTV cameras provide a video image of roadways at a variety of locations throughout a large geographic area.

In our demonstration applications, we also use probe vehicles. Probe vehicles are geolocated in real time to provide second-by-second tracking of the vehicle fleet. This type

of fleet management is done by both the public and private sectors (e.g., transit management or delivery fleet management).

In the next two sections we describe our instantiation of servers from top to bottom, which is the direction of data flow in Figure 19. The parallelism between the server structures to support the applications described is clearly visible in Figure 19, with TrafNet on the left and BusView on the right.

6.4.1 TrafNet

We use two types of data sources to build a variety of applications using our development environment. The first example is an application we call TrafNet, which provides real-time updates of traffic conditions to Internet users.[1] In Figure 19, the left side of the diagram represents the architecture of TrafNet, and the data source is a Traffic Management System (TMS) operated by the Washington State Department of Transportation (WSDOT), which polls several thousand inductance loops every 20 seconds. WSDOT, which funds the collection of these data, uses the data for traffic management; however, this type of information has potential value to travelers and shippers as well. We make these data, and products derived from them, available on the Internet using the paradigms proposed here.

The Source component in this application eavesdrops on the loop data while the loops are being polled and maintains a data dictionary that describes the structure of the loop data. This structure includes information about loop locations and loop type (e.g., the type of measurement and units of the measurement). The Source is connected to a network and responds to requests for loop data by first sending the data dictionary and then following that with the updated set of sensor values as soon as they become available. This process is actually implemented on the computer that runs the WSDOT TMS system and is named TMSUW.

---

[1] The executable for *TrafNet* can be obtained from http://www.ivhs.washington.edu/trafnet.

The next component, labeled Loops_forward, is an Operator that receives the data dictionary and data from TMSUW. To these data it adds information about the route and milepost marker locations for each of the items found in the data dictionary, and, thus, this Operator component is engaged in "data fusion." This component operates on a computer that is located at WSDOT's Traffic Systems Management Center (TSMC), and the significance of the physical location is that this data fusion server can control what data are

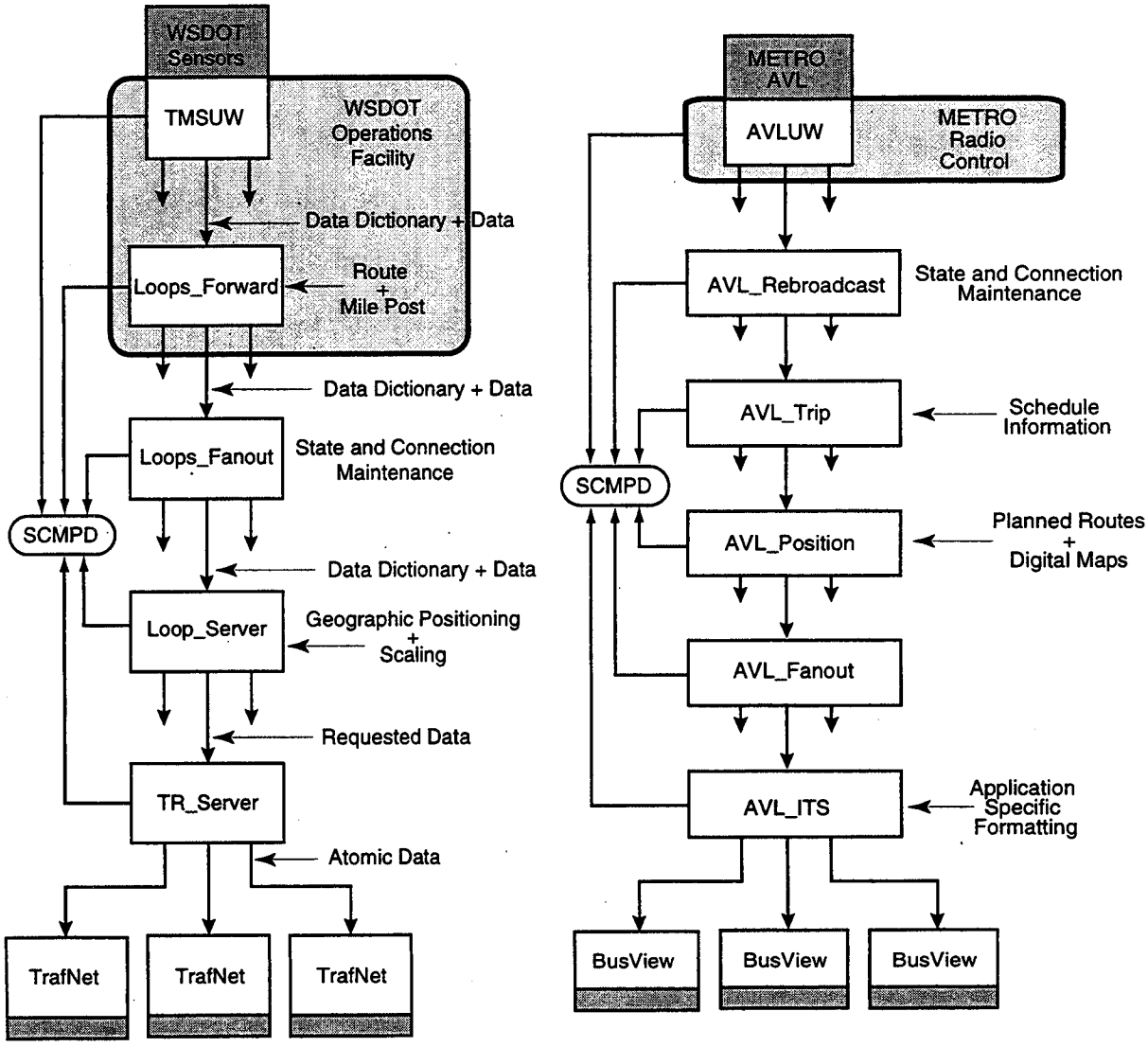


Figure 19. TrafNet and BusView applications

allowed (by agreement with WSDOT staff) to leave WSDOT's premises. This control over access to the data provides agency autonomy and network security. There are two firewalls between the TSMC subnet and the Internet that protect WSDOT internal computers from the Internet community. The firewalls allow only certain computers at specific TCP ports to connect to this computer.

The component labeled Loop_Fanout in Figure 19 is a Redistributor component that makes the loop data and data dictionary available to a larger number of clients. It maintains connectivity with Loops_forward while accepting and authenticating requests for data connections. This is the last component in the hierarchy where the original data as recorded by the sensors are available; downstream servers modify the original data stream.

The component labeled Loop_Server is an Operator that fuses data by adding geoposition in latitude and longitude to the data dictionary on the basis of the loop identifier and the mile post information. It further scales the data to known units (e.g., vehicle counts converted to vehicles per hour).

The next server component, labeled TRServer, is another Operator component that combines data from individual loops to make congestion estimates. It then packages the resulting information in a specific format and order to be used by the presentation Sink component labeled TrafNet. The connection between this server and the TrafNet Sink can potentially employ slower media, so this component minimizes the amount of data that need to be sent per update of the TrafNet screen.

To make the TrafNet data stream available to large numbers of users, we can add Redistributors downstream of this component. However, to date, that has not been necessary.

The TrafNet program (the Sink at the bottom) is a Microsoft Windows 3.1 application that uses the Winsock network interface. The end user looking at the TrafNet screen sees only the graphic representation of the freeway system with congestion indications updated every 20 seconds. The user can select entrance and exit ramps to

request information about specific trips. This is the first example of how our approach facilitates the creation of value-added applications and also makes intermediate forms of the data available for other uses.

6.4.2 BusView

A second example of a distributed application created within our framework is BusView, which uses probe vehicles - in this case transit coaches - to create an Advanced Traveler Information System (ATIS) application that displays the location of the transit coaches on a digital map in real time.

BusView, shown in the right-hand column of Figure 19, uses a transit carrier's automatic vehicle location system as the sensor for the Source component. In this case, the Source component (labeled AVLUW) eavesdrops on communication between components of a proprietary AVL system used by the transit carrier's operations staff to measure schedule adherence and to provide traffic managers geolocation specifics in the event of an incident. The data, in this case, constitute distance along a planned path, which, with knowledge of the routes and several coordinate transformations, can be used to estimate geolocation.

Like the component Loops_forward in the previous example, AVL_AVLUW is an operator located on a computer in the AVL operations center of the transit carrier, and, like the previous example, it limits the information that can be sent out of the agency. In this case, the numerical identification of the driver is in the data received by AVLUW but is removed for privacy reasons before the data are allowed out of the physical control of the transit agency. This component is also behind firewalls that protect the transit AVL system from Internet users. Once again, this Operator component provides for network security and agency autonomy, while also removing information from the data stream.

The raw data, a distance along a known path, is multiplexed by a Redistributor (labeled AVL_Rebroadcast). Like the component Loops_fanout in the previous example, AVL_Rebroadcast multiplexes the original raw data to make them widely available.

The data from AVL_Rebroadcast are passed to an Operator (labeled AVL_Trip), which uses information in the AVL data packet to associate a specific vehicle with a specific trip in the transit schedule. The data from the AVL system arrives at an average rate of 11-coaches-per-second, and a 20,000 record database of scheduled trips is searched for each coach to obtain the schedule correspondence. This trip information is added to the record for each coach and made available to downstream clients.

The data from AVL_Trip is passed to an Operator (labeled AVL_Position), which can use the data, information about planned routes, and digital maps, and then coordinate transformations to calculate a latitude and longitude value for the transit (probe) vehicles. The calculation of geoposition requires (1) identifying the particular coordinate lists in one of 2500+ files that represent every possible vehicle route, (2) selecting the segment within the list on which the vehicle is estimated to exist, and (3) using a Newton's method to perform an inverse Lambert projection from the proprietary coordinate system of the AVL system at a rate of 11-vehicles-per-second. The ability of a component to operate on independent CPUs, which is designed into our approach, makes it possible to select a CPU of appropriate power for each of the designated tasks. This Operator, for example, takes the entire resources of one computer to keep up with the data flow, so it does not coexist on a CPU with the other Operators mentioned in these examples.

Downstream of the AVL_Position server is a Redistributor labeled AVL_Fanout which multiplexes the data stream containing vehicle geopositions to a variety of users. This Redistributor frees the upstream process from multiplexing so that it can focus on determining the positioning solution.

The server component labeled AVL_Ivhs creates a customized data stream for the BusView presentation. Like TRServer in the previous example, the connection between this server and the BusView Sink can potentially employ slower media. This component

minimizes the amount of data that needs to be sent, per update of each transit vehicle, on the BusView screen.[2]

A digital map and vehicle location information, along with schedule information, are displayed graphically on the BusView screen, creating an information system for transit riders. BusView is a second example of how our approach can be applied to create distributed applications that use real-time data from a single source. In the next section, we describe a hybrid system that takes data from both of the server stacks that have just been described.

6.4.3 Hybrid Example: SWIFT

A third instantiation of our paradigms is a hybrid of the previous two applications. This hybrid application is known as Seattle Wide-area Information for Travelers (SWIFT), an operational test sponsored by the Federal Highway Administration (FHWA). This project combines data from the sources used for both TrafNet and BusView to create new information. Traffic congestion and probe vehicle information described above are used to estimate speed (and travel times) on freeway links, and that information, in turn, is provided to a commercial wireless carrier, which broadcasts it to subscribers. These subscribers receive the information via three principle devices: (1) a Seiko Receptor ™ watch, (2) a Delco car radio, and (3) an IBM portable computer. Each of the receiving platforms represents a potential value-added resale of the information. Figure 20 shows the collection of components that create the information source for the SWIFT application. Note that many of the components that were part of the other two applications are also used in the SWIFT application, and this simultaneous usage does not affect the operation of the other applications.

---

[2] Request for access to the BusView URL can be sent to busview@ivhs.washington.edu.

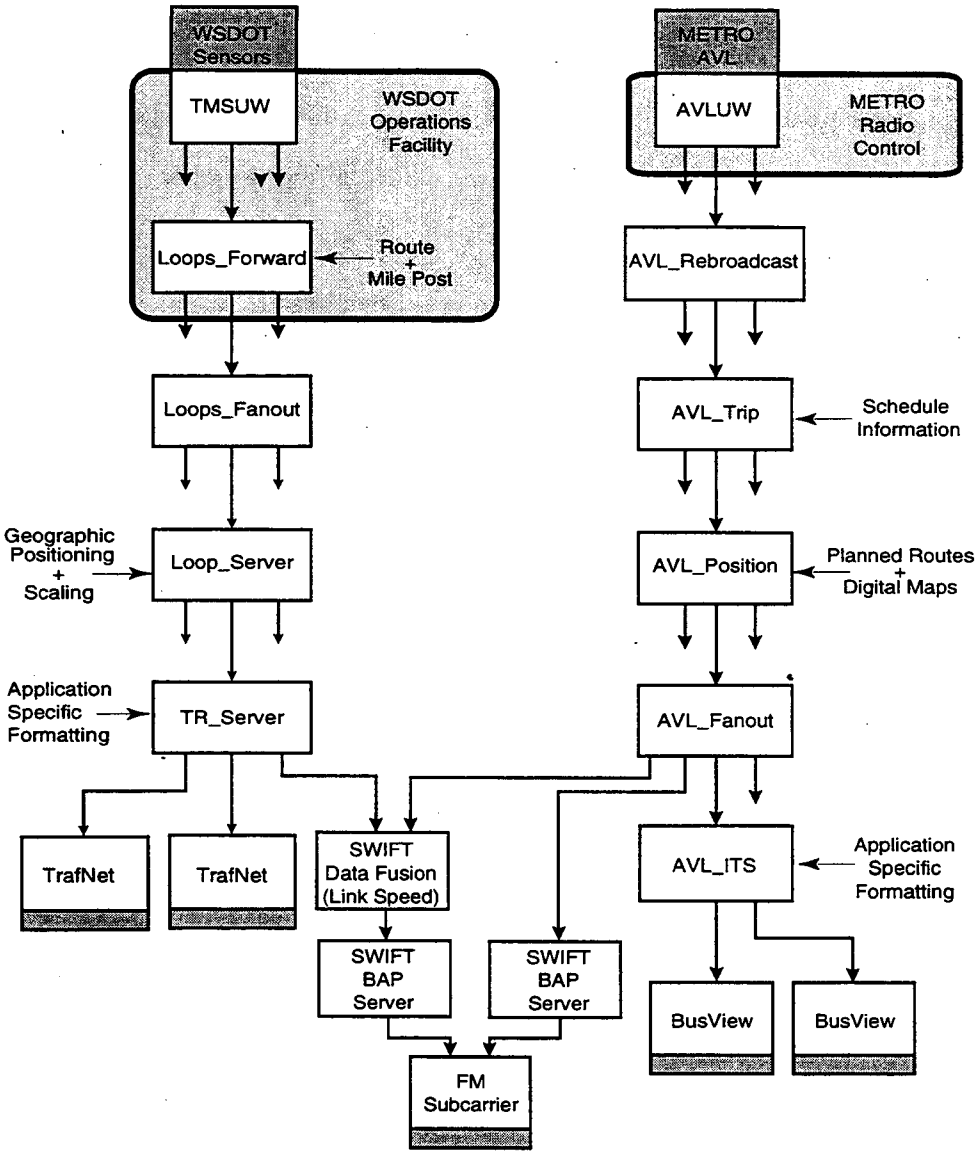


Figure 20. SWIFT application in the context of TrafNet and BusView

7. CONCLUSION

The approach described in this paper embodies several specific features, which include (1) an architecture that will deliver real-time data to a large number of consumers while maintaining a mechanism for authorization; (2) a definition of distributed applications that includes an entire set of components arranged in an hierarchical structure; and (3) a clear mechanism for building "value-added" applications that use a generally available data stream. In this report, we have described in detail a set of paradigms to implement such distributed applications and have discussed several actual implementations in the domain of ITS.

We have presented a conceptual framework for ITS development and have shown how this framework solves numerous high-level problems associated with ITS development. A sample instantiation, the Backbone project, further demonstrates the viability of our unified ITS conceptual framework and shows that such a framework can be implemented at reasonable cost and with a high likelihood of successful operation.

REFERENCES

Albus, James S., Maris Juberts, and Sandor Szabo. A Reference Model Architecture for Intelligent Vehicle and Highway Systems. *Proceedings of the Intelligent Vehicles '92 Symposium,* Detroit, Michigan, June 29-July 1, 1992, pp. 378-84.

Chadwick, Jim, and Vijay Patel. A Communications Architecture Concept for Intelligent Vehicle Highway Systems (IVHS). *Proceedings of the Intelligent Vehicles '92 Symposium*, Detroit, Michigan, June 29-July 1, 1992, pp. 359-64.

Chen, Kan, and Bernard Galler. An Overview of Intelligent Vehicle-Highway Systems (IVHS) Activities in North America. *Proceedings of the 5th Jerusalem Conference on Information Technology (JCIT)*, Jerusalem, Israel, October 1990, pp. 694-701.

Hsin, Victor J. and Paul T. R. Wang. Modeling Concepts for Intelligent Vehicle Highway Systems (IVHS) Applications. *1992 Winter Simulation Conference Proceedings*, Arlington, Virginia, December 13-16, 1992, pp. 1201-09.

Kramer, J., J. Magee, M. Sloman, and N. Dulay. Configuring Object-Based Distributed Systems in REX. *IEEE Software Engineering Journal*, 7(2):139-49, March 1992.

Kramer, Jeff, Jeff Magee, and Keng Ng. Graphical Configuration Programming. *IEEE Computer*, 22(10): 53-8, 62-5, October 1989.

Laraqui, Kim, Magnus Lengdell, Frank Reichert, and Adreas Fasbender. Implementation of Communication and Management Protocols for the Integrated Road Transport and Traffic Environment. *1994 IEEE 44th Vehicular Technology Conference,*. Stockholm, Sweden, June 8-10, 1994, pp. 379-83, vol. 1.

Magee, Jeff, Naranker Dulay, and Jeff Kramer. Regis: A Constructive Development Environment for Distributed Programs. *Distributed Systems Engineering Journal*, 1(5):304-12, September 1994.

Magee, J., Jeff Kramer, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6): 663-75, June 1989.

Purtilo, James M. The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151-74, January 1994.

Rockwell International Joint Architecture Team, Loral Federal Systems. Physical architecture. *ITS Architecture*, October 1995.

Rockwell International Joint Architecture Team, Loral Federal Systems. Theory of operations. *ITS Architecture*, October 1995.

Schill, A. Distributed Application Support: Survey and Synthesis of Existing Approaches. *Information and Software Technology*, 32(8):545-58, October 1990.

Varaiya, Pravin, and Steven Shladover. Sketch of an IVHS Systems Architecture. *Vehicle Navigation and Information Systems Conference Proceedings*, Dearborn, Michigan, October 1991, pp. 909-22, vol. 2.