Title
Middleware for Cooperative Vehicle-Infrastructure Systems

Authors
Manasseh, Christian
Sengupta, Raja

Publication Date
2008

CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Middleware for Cooperative Vehicle-Infrastructure Systems

Christian Manasseh, Raja Sengupta

California PATH Research Report
UCB-ITS-PRR-2008-2

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation, and the United States Department of Transportation, Federal Highway Administration.

Final Report for RTA 65A0177-15650

January 2008

ISSN 1055-1425

CALIFORNIA PARTNERS FOR ADVANCED TRANSIT AND HIGHWAYS

Middleware for Cooperative Vehicle-Infrastructure Systems

RTA#65A0177-15650

Christian Manasseh & Raja Sengupta

11/13/2007

Middleware for Cooperative Vehicle-Infrastructure Systems

Christian Manasseh & Raja Sengupta

Nov. 13, 2007

Abstract

Middleware has emerged as an important architectural component in supporting distributed applications. The role of middleware is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution. Mobile sensors fall into the space of distributed systems that suffer from isolated data sources, heterogeneous communication infrastructure and varying application requirements. In this report, we provide a middleware architecture that addresses the needs of a distributed system made of mobile sensors in general and discuss the implementation of this middleware architecture in a mobile sensor network comprised of vehicles and intersections producing traffic related data for traffic safety and operations. We conclude our report with some performance measures that relate to the cost of overhead incurred from using the middleware which prove it efficient for traffic management applications.

Keywords: Advanced Driver Information Systems, Advanced Traffic Management Systems, Advanced Vehicle Control Systems, Architecture, Data Communication, In-vehicle Information Systems, In-vehicle Sensing Systems, Intelligent Vehicle Highway Systems, Real-time Information, Software Engineering, System Architecture

Executive Summary

Middleware has emerged as an important architectural component in supporting distributed applications. The role of middleware is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution. The importance of the topic is reflected in the increasing visibility of standardization activities such as the ISO/ITU-T Reference Model for Open Distributed Processing (RM-ODP), OMG's CORBA, the Java RMI, Microsoft's .NET and the Open Group's DCE. Mobile sensors fall into the space of distributed systems that suffer from isolated data sources, heterogeneous communication infrastructure and varying application requirements. In this report, we provide a middleware architecture that addresses the needs of a distributed system made of mobile sensors. We then discuss the implementation of this middleware architecture in a mobile sensor network comprised of vehicles and intersections producing traffic related data for traffic safety and operations. Finally, we present performance measures that relate to the cost of overhead from using the middleware.

Mobile sensor networks have a variety of applications. Examples include traffic monitoring which involves vehicle driver and infrastructure, environmental monitoring which involves monitoring air soil and water, condition based maintenance, seismic detection, military surveillance, inventory tracking, etc. In fact, due to the pervasive nature of micro-sensors and the widespread use of wireless communication, mobile sensor networks have the potential to revolutionize the way we understand and construct complex physical systems (Estrin, et al. 1999). Data generated from these sensors varies by type, quality and quantity within the same application; the data can travel over a wide range of wireless communication media such as WiFi, WiMAX, GPRS, DSRC, etc. and it can serve multiple purposes depending on the user needs. An auto-service technician might be interested in the engine reading for a specific vehicle as it travels on a section of the roadway and connects to the service warehouse over GPRS; whereas, a traffic operations engineer would be interested in the average speed of the thousands of vehicles that cross a certain point on the highway during a certain time period and communicate over DSRC. In this report we present a middleware layer that interfaces with the hardware (sensors and communication media) from one side and presents a single interface to the application developer (software) on the other side.

The middleware offers three main simplifications to the complexity of mobile sensor network applications. First it reduces the complexities of the heterogeneous communication layer and data sources to a simple standardized *services* interface allowing the application developer to consume whatever data is generated from the sensors. In our implementation we choose web services as the middleware interface. Second, it solves sensor discovery problems by providing a geo-spatial query interface allowing the data consumer to locate the sensors of choice by providing a relational query engine. And third, it offers seamless connectivity between the application (data consumer) and the sensors (data source) by hiding the intricate details of the several communication standards from the application developer. In our implementation we utilize the concept of a message broker to develop the geo-spatial discovery and seamless connectivity features of the middleware. As a result, the application developer's task is reduced to just connecting to the middleware *services* interface and not having to worry about varying communication standards, different hardware types and firmware versions on the sensors, different representations of sensor data, etc.

Table of Contents

List of Figures

Middleware for Cooperative Vehicle-Infrastructure Systems

Introduction

In the US, EU and Japan very substantial programs exist for the development and deployment of cooperative-vehicle infrastructure systems. In the US, Vehicle Infrastructure Integration (VII) is an initiative fostering research and applications development for a series of technologies directly linking road vehicles to their physical surroundings, first and foremost in order to improve road safety as well as traffic efficiency (US Department of Transportation 2007). At the EU level, the Cooperative Vehicle-Infrastructure Systems (CVIS) underlines the importance of intelligence in-car and roadside systems for improving traffic safety and efficiency and environmental impact (Reding 2006). In Japan, the Smartway Project, a national level project, enables communication among vehicle, driver and pedestrian with advanced ITS technologies (Setsuo 2007). In this report we will briefly present each of those systems and then present a different approach based on a service-oriented middleware for handling cooperative vehicle-infrastructure systems.

Middleware has emerged as an important architectural component in supporting distributed applications. The role of middleware is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution. The importance of the topic is reflected in the increasing visibility of standardization activities such as the ISO/ITU-T Reference Model for Open Distributed Processing (RM-ODP) (Tindale-Biscoe 2002), OMG's CORBA (Raj 1998), the Java RMI (Raj 1998), Microsoft's .NET and the Open Group's DCE. Traffic safety and traffic monitoring can be considered among the distributed applications that can benefit from a middleware-based architecture. Sensor-equipped vehicles and traffic control elements (traffic signals, changeable message signs, loop detectors, etc.) fall into the space of distributed systems that suffer from isolated data sources, heterogeneous communication infrastructure and varying application requirements. Data generated from these sensors varies by type, quality and quantity within the same application; the data can travel over a wide range of wireless communication media such as WiFi, WiMAX, GPRS, DSRC, etc. and it can serve multiple purposes depending on the user needs. A traffic operations engineer would be interested in the average speed of the thousands of vehicles that cross a certain point on the highway during a certain time period and communicate over DSRC; whereas, an auto-service technician might be interested in the engine reading for a specific vehicle as it travels on a section of the roadway and connects to the service warehouse over GPRS.

The middleware offers three main simplifications to the complexity of mobile sensor in a vehicle trying to communicate with the infrastructure. First it reduces the complexities of the heterogeneous communication layer and data sources to a simple standardized *services* interface allowing the application developer to consume whatever data is generated from the sensors. In our implementation we choose web services as the middleware interface. Second, it solves sensor discovery problems by providing a geo-spatial query interface allowing the data consumer to locate the sensors of choice by providing a relational query engine. And third, it offers

seamless connectivity between the application (data consumer) and the sensors (data source) by hiding the intricate details of the several communication standards from the application developer. In our implementation we utilize the concept of a message broker to develop the geo-spatial discovery and seamless connectivity features of the middleware. As a result, the application developer's task is reduced to just connecting to the middleware *services* interface and not having to worry about varying communication standards, different hardware types and firmware versions on the sensors, different representations of sensor data, etc.

This report is structured as follows: we first present the various approaches that are being taken by several government and private sector entities in the field of cooperative-vehicle infrastructure systems, then we present the service-oriented middleware architecture for interconnecting vehicle and infrastructure sensors, in this section we introduce service-oriented architectures and component-based programming models. In the final section of the paper we discuss the details of our implementation of a service-oriented middleware architecture for cooperative vehicle infrastructure systems.

Current efforts for cooperative-vehicle infrastructure systems

The US VII Program

The VII Program concept is an outcome of the foreseeable coordinated deployments of communication technology in all vehicles by the automotive industry and on all major U.S. roadways by the transportation public sector. The U.S. DOT perceives the possibility of major improvements in highway safety through crash prevention. The U.S. DOT also envisions VII enabling more effective operation of the state and local transportation systems through collection of valuable information about the real-time status of the roadways (ITS U.S. DOT 2006). The VII coalition is a cooperative venture of the U.S. DOT, the automotive industry, American Association of State Highway and Transportation Officials (AASHTO) and State DOTs. Some of the preliminary actions taken by the government to induce the VII Program involve the Federal Communications Commission (FCC) and the standards development organizations licensing and establishing a vehicle-to-vehicle and vehicle-to-roadside communications system over Dedicated Short Range Communications (DSRC) at 5.9 GHz (ITS U.S. DOT 2006). Vehicle Infrastructure Integration (VII) applications involve the harvesting of data from different types of sources with different characteristics. The data of interest represents properties of vehicle, driver and/or infrastructure control elements. The source of data can be the vehicle On-Board Diagnostic (OBD) systems, add-on vehicle sensors that detect vehicle and/or driver properties, infrastructure intrusive (loop detectors) or non-intrusive sensors (cameras, radar), control elements computer systems such as traffic light controllers(170, 2070), as well as other sources of sensors and systems that can portray the characteristics and behavior of the traffic stream or need to convey information (safety-related or otherwise) to the driver. The data itself can be in the form of messages or files of varying sizes and of varying targeted audiences. Furthermore, this data has a varying life span; some might be useful for the few milliseconds or seconds in which it originates, and some might be useful in an archived manner that would span over several years. The Vehicle Safety Communications (VSC) project which has the U.S. DOT and several automakers on its list of members, has analyzed several application scenarios, 43 of which 33 are in safety and 10 in non-safety applications will be referenced in this research (The

CAMP Vehicle Safety Communications Consortium 2005). In our research we will rely on those scenarios as the major requirements for implementing software applications in VII.

The U.S. DOT, through its ITS Joint Program Office, has come up with a VII Architecture document that defines the several components and functional elements of a VII deployment (ITS Joint Program Office 2005). The following is a quick description of the components that are of interest to our research:

Road-side unit (RSU): the road-side equipment that would be part of the network infrastructure, providing a communication hot spot, storage, internet connectivity and in some cases information about a road side control unit such as a traffic light, changeable message sign (CMS), pavement senor-read data, etc. Figure 1**Error! Reference source not found.** shows the usual components that compose the RSU.

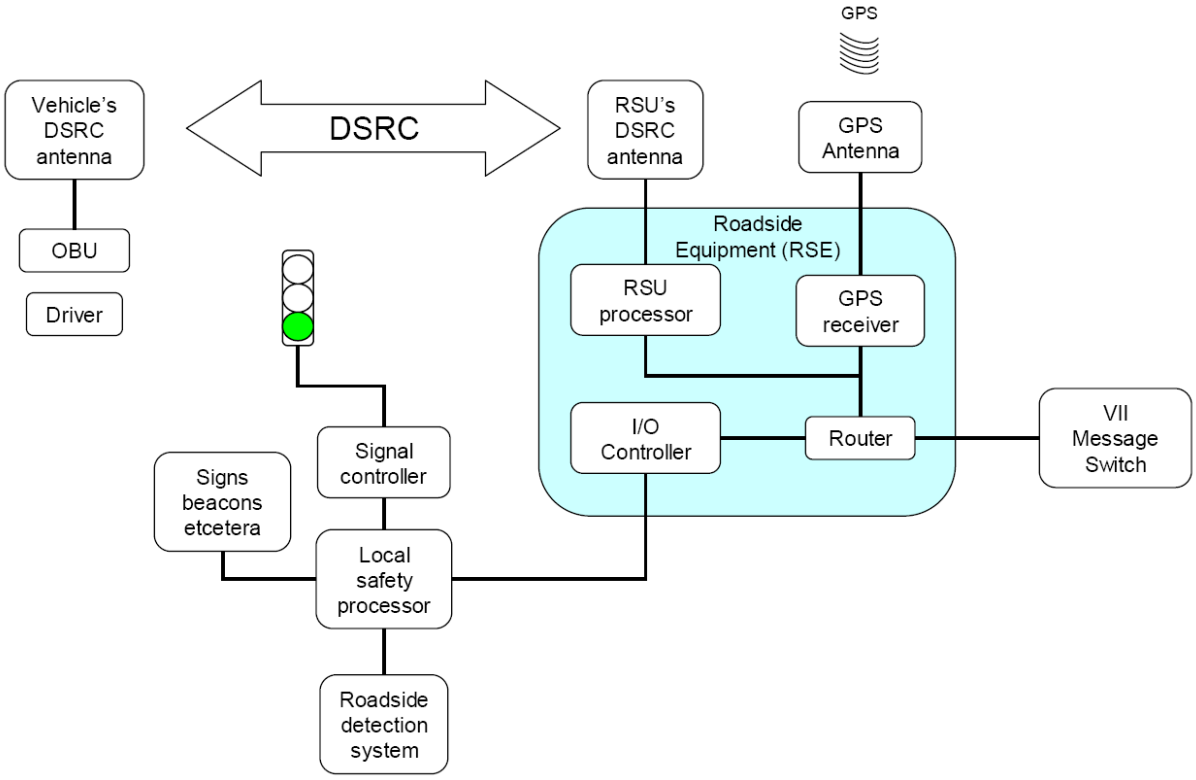


Figure 1. RSU Logical Layout (ITS Joint Program Office 2005)

The Roadside Equipment (RSE) is composed of the RSU processor which is in many cases a regular computer processor, a GPS receiver to identify position, an Input/Output (I/O) Controller that communicates with the several hardware that might exist at an RSU (such as an intersection controller or Changeable Message Sign (CMS) processor), and a router to connect to in internet backhaul network. The GPS receiver is connected to a GPS antenna to receive GPS data about location. The RSE is has a DSRC antenna allowing it to communicate to vehicles in the DSRC range.

On-Board Unit (OBU): this is the equipment installed inside the vehicle and interfaces with the vehicle diagnostic board. Data from the vehicle about the vehicle and driver behavior could be

relayed using this unit to the outside world (relative to the vehicle) and could also be stored in the local storage component of this unit. Figure 2 shows the usual components that compose the OBU.

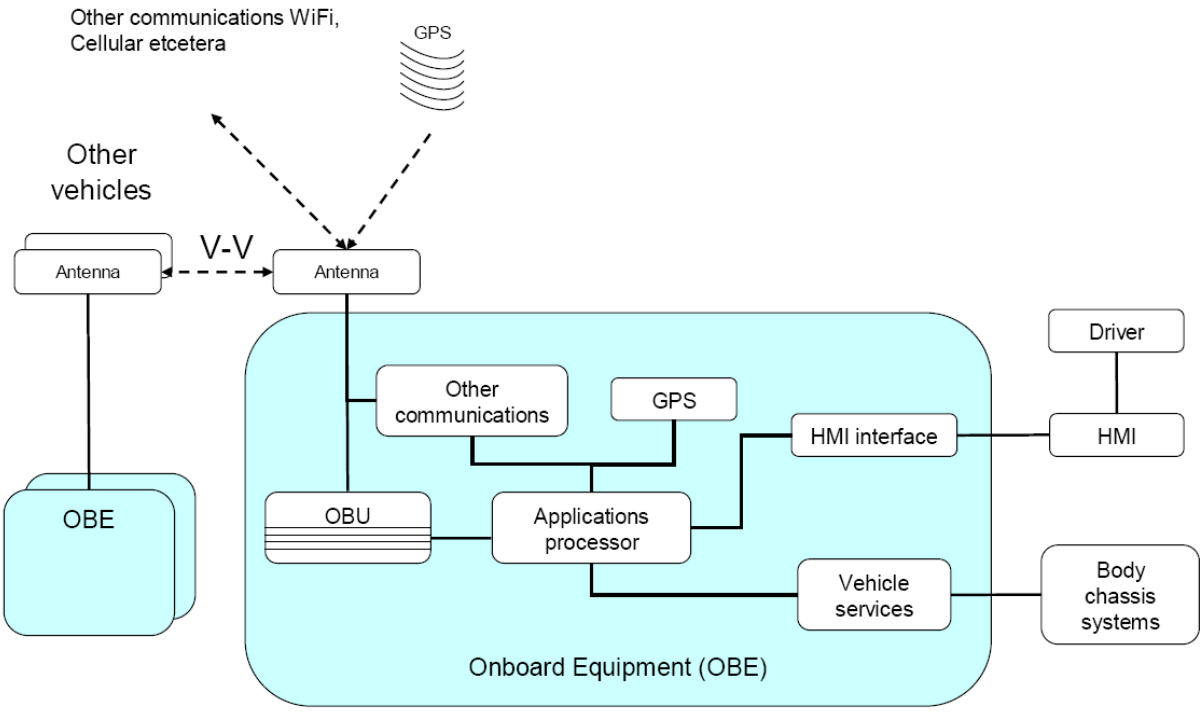


Figure 2. OBU Logical Layout (ITS Joint Program Office 2005)

The body chassis systems are a set of sensors feeding into the OBD connector of the vehicle. Those would offer their data as vehicle services to the OBU processor, which is in many cases a simple computer processor. The Human-Machine Interface (HMI) provides the interface to the driver. GPS receiver and antenna provide location of vehicle. Communication devices that rely on DSRC, WiFi, Cellular, etc will also be made available in the OBU.

Dedicated Short Range Communication (DSRC) layer: the DSRC is the government licensed bandwidth provided for the use of VII application. It is beyond the scope of this report to address the many facets of this technology, but it will be sufficient to say that this research will utilize the bandwidth and the different network layers provided by this technology to relay messages between the different components of the network. As of yet, no standardized messaging protocol has been issued for communicating on this bandwidth and it is the purpose of this research to provide a general enough architecture model to allow any type of message to be transmitted on the DSRC communication layer.

The U.S. DOT VII Architecture

By using the VII components presented earlier, the U.S. DOT has put together the proposed VII Architecture for message flow between vehicles and infrastructure elements.

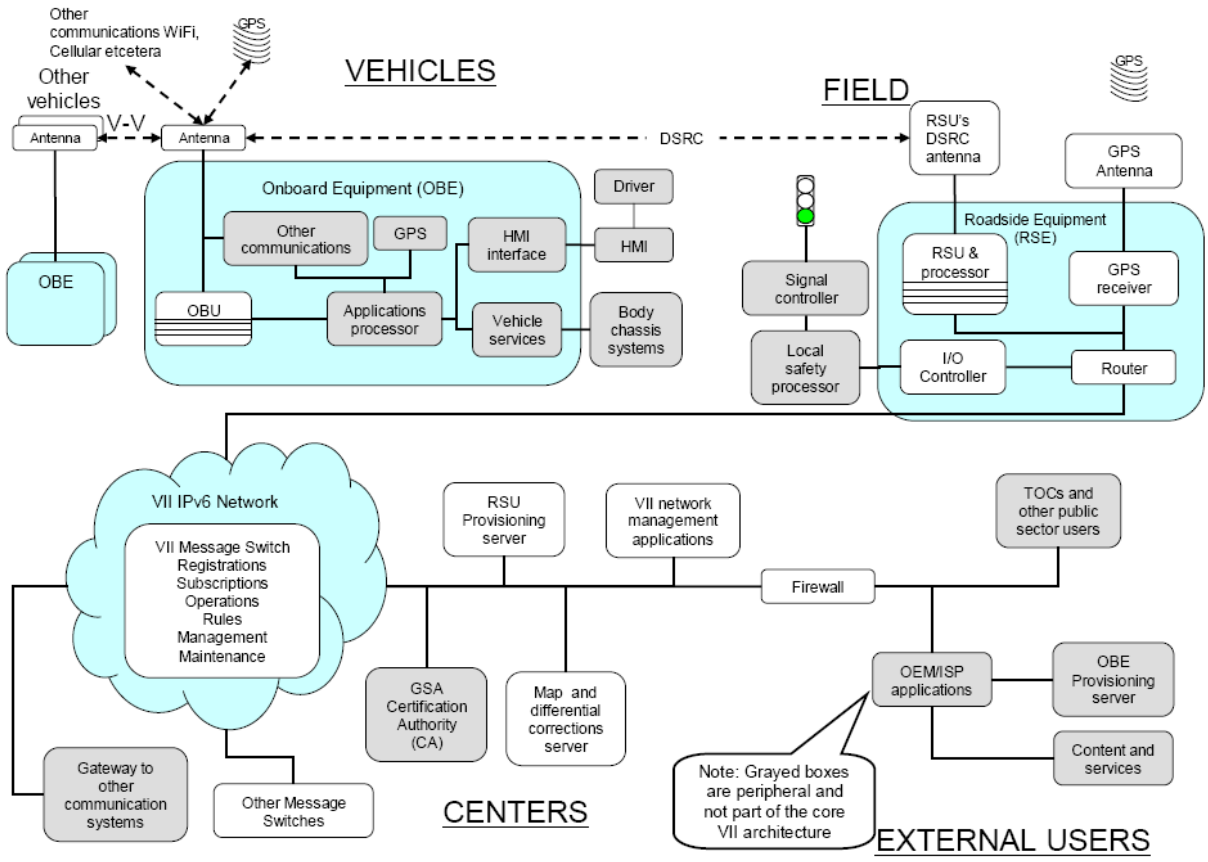


Figure 3. U.S. DOT VII Architecture (ITS Joint Program Office 2005)

Figure 3 is a simplified view of this architecture that shows the basic elements of a VII Application. According to this architecture, vehicles would communicate with each other (V-V) using DSRC, Cellular, WiFi or other communication media; vehicles would communicate with field RSU's using DSRC. The RSU would connect to the VII Message switch through its backhaul connection. Any user or center wishing to consume the data being gathered by the RSU's would have to connect via the VII Message Switch.

The VII Message Switch is designed to operate in a publish-subscribe method, by which RSU's would publish their data to the message switch inbox and any users, centers or applications wishing to make use of that data have to subscribe with the VII Message Switch. The U.S. DOT envisions a centralized architecture for the VII Message Switch by which there will be two main VII Message Switches to cover the continental United States: one for the West Coast and one for the East Coast.

The EU CVIS Program

The European ITS project CVIS will bring major functional improvements to road users by allowing vehicles to communicate and cooperate directly with others nearby and with the roadside infrastructure (Mietzner 2007). CVIS has underway a prototype for a platform providing a wide range of functionality for journey support, information and security services offered to road operators and drivers.

The CVIS technology for vehicle-to-vehicle (V2V) and vehicle-to infrastructure (V2I) communication, is based on a multi-channel terminal capable of connecting to a wide range of potential carriers, including WLAN/Wi-Fi, Cellular (GPRS, UMTS), DSRC, and Infra-red (IR). This is based on the new international Continuous Air interface for Long and Medium range (CALM) standard (Figure 4) which will provide full interoperability between different car brands and different roadside and infrastructure systems. It is important to mention that the CALM specifications/standards are not a physical piece of equipment. While CALM may indeed operate from a "box" designed to achieve its functions, CALM is actually a related set of protocols, procedures and management processes (ISO 2007). While it may appear in the vehicle in the form of a box, it is just as likely to be incorporated into one of the in-car computing functions. CVIS/CALM equipped vehicles will be able to connect and communicate via local ad-hoc networks with vehicles and roadside equipment in the vicinity and also with an internet connection.

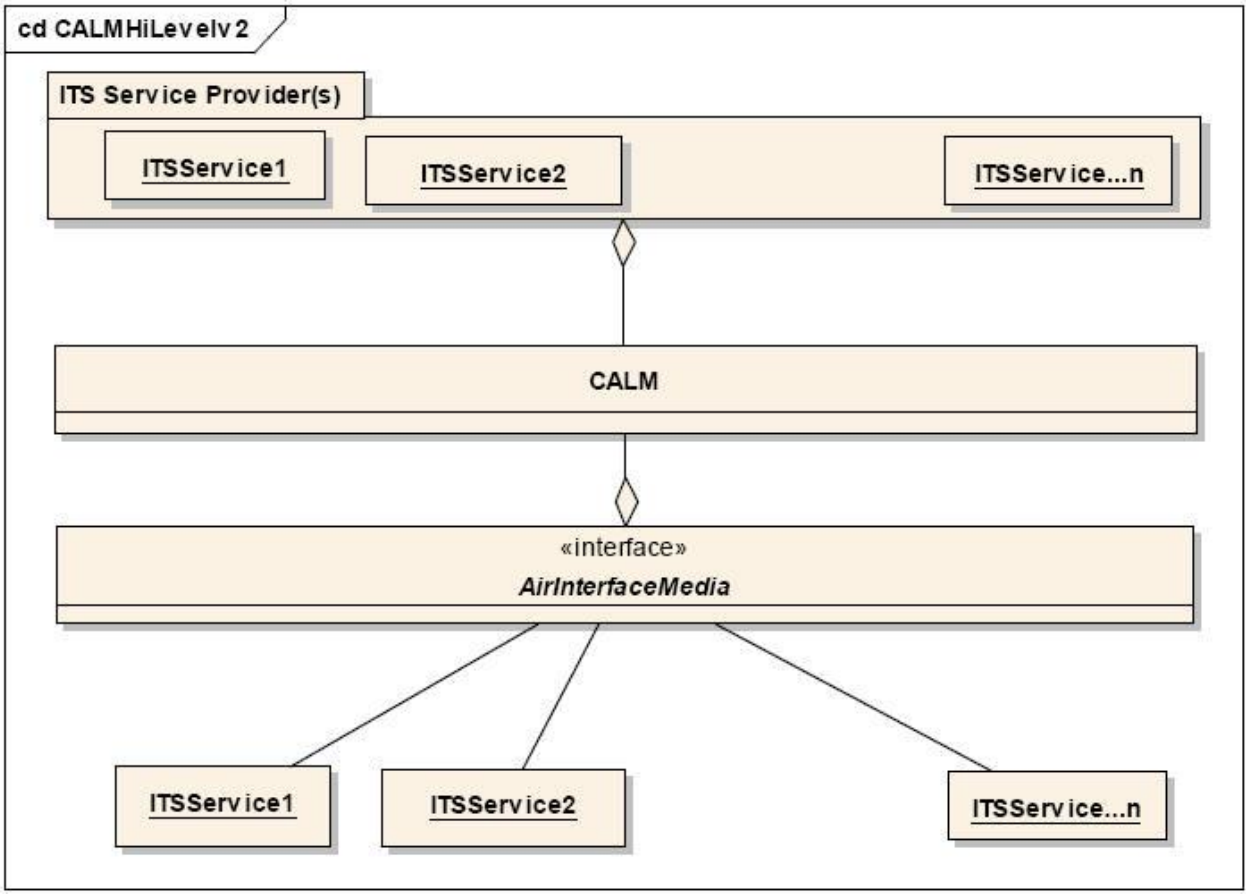


Figure 4. The CALM Concept at the highest level of abstraction (ISO 2007)

Similar to the 43 application scenarios outlined by the US VII Program, CVIS outlined a number of sample applications for various types of road operators. These applications are related to (Mietzner 2007):

- Destination route guidance
- Congestion control and avoidance

- Traffic lights control and coordinating
- Area-wide Traffic Information Provisioning (traffic speed, congestion “hot spots”, road conditions etc.)
- Truck and vehicle monitoring and (micro) management
- Parking and loading space management
- Safety information, collected from rain, fog ice and air quality sensors in vehicles and road side units.

The CVIS architecture, while it contains similar elements to the VII RSU, and OBU it tries to address a more challenging set of requirements such as providing a transparent V2V and V2I communication layer that requires no application setup and management; confirm to modern internet techniques and standards for global usability while adhering to the wide range of different possibilities related to data speeds, communication distance, cost and other parameters. Another important factor in the CVIS architecture is the capability of organizing the system in a decentralized manner. The decentralized element of the CVIS architecture is handled by a set of sub-projects such the COMO and FOAM. Figure 5 below outlines the various sub-projects under the CVIS project umbrella. We highlight the COMO and FOAM projects as the two main projects related to the work we discuss in this report

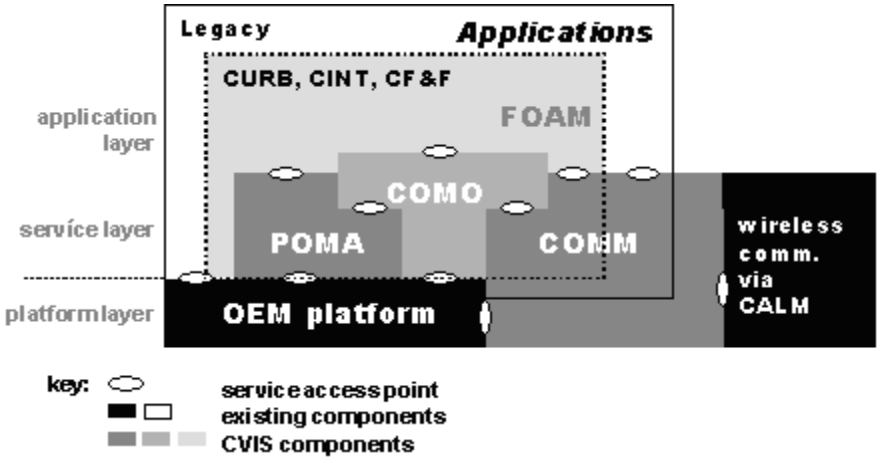


Figure 5. The CVIS Projects Architecture (CVIS 2007)

The COMO sub-project aims to develop specifications and prototypes for the collection, integration and delivery of extended real-time information on individual and collective vehicle movements and on the state of the road network. Its main purpose is for collaborative monitoring of vehicle movement (CVIS 2007). The COMO Sub-project is placed as a central basic service inside the CVIS framework. COMO will cooperate closely with the application oriented activity blocks for urban (CURB), inter-urban (CINT) and fleet & freight (CF&F) applications to capture their particular requirements about monitoring of traffic and environmental information. The services are based on a consistent set of interfaces, which is applicable throughout the entire CVIS platform to control and manage the required transactions and data flows between vehicles, roadside infrastructure and back office systems like service or traffic management centers. In order to ensure the cooperation between these three entities COMO will provide distributed and decentralized data processing components.

CVIS will potentially capture far more data, and from more vehicles (since all vehicles will be able – through use of standard-based onboard units – to contribute data, not just those subscribing to a specific service). While this will provide the highest quality information on the real-time status over the entire road network, it will also lead to an enormous growth in data communications volume (CVIS 2007). A major challenge therefore for COMO is to create a distributed architecture where a maximum of processing can be done locally, in individual or amongst groups of vehicles, and where the volume of data transmissions can be moderated according to the context and content of the data (so that more data are transmitted that refer to an important exception – e.g. traffic incident – than for non-changing or slowly changing traffic status). The COMO services will be implemented on the FOAM platform and integrated in the local target environment.

FOAM will define an architecture that connects the in-vehicle systems, roadside infrastructure and back-end infrastructure that is necessary for co-operative transport management. The aim is to produce an architecture and specification that is implementation-independent, i.e. allows different implementations for various client and back-end server technologies. The features of FOAM aims are (CVIS 2007):

- exchanging and updating of service application components at any domain in the CVIS architecture
- product / vendor independence by adaptable middleware components
- common design of structural elements (e.g. data exchange formats, protocol specifications, API's and run-time environment)
- common design of secure communication in distributed systems including billing, authentication of user data and authorization of users
- re-usability of components by generalized access of resources.

The Japanese Smartway Project

In Japan, the Smartway Project positioned as a national level project, enables communication among vehicle, driver and pedestrian with advanced ITS technologies. The Smartway project strategy is focused on reducing deaths at the roads (Setsuo 2007). The Smartway architectural components are similar to the US and EU-based programs in which a vehicular OBU and a roadside RSU are required. Although the Smartway project has not progressed as much as its counterparts in the US and Europe it does pose similar requirements that will face the same challenges. The architecture of this project is not yet available; however, several advanced ITS efforts are being demonstrated for the purpose of funneling in the Smartway project.

From what has preceded in the description of each of the programs, we realize that the challenge of connecting the data sources, be it vehicle sensors or road side units, is being approached from a tight coupling approach between the hardware and the software layers. In the US program, this tight coupling is clearly obvious by the nature of the communication standards and the requirements driving the program, the EU project seems to be more open to a more loosely coupled architecture. However, when looking at the details of how the FOAM and COMO layers of the CVIS architecture interface with each other and the other elements of the system; we realize that the interface is well defined for a certain set of hardware/communication

combination. The use of interface API in FOAM make the architecture more open than the US VII, but does not offer the loose coupling that we propose in our service-oriented architecture.

In order to better understand what we mean by loose coupling and service-oriented architecture, the following section will provide a quick overview of component-based technologies and the evolution of software programming paradigms to arrive at what we now call service-oriented architectures (SOA).

Component-based Technologies

Having listed the various endeavors in building out vehicle-infrastructure integration, we introduce the field of component-based technologies that currently exists in the software development industry from which we will develop our perception of interconnecting the vehicle and infrastructure. We also provide a brief history of how software architecture evolved to rely on services.

Software Architecture Evolution

Historically, software programs were written to automate hardware. In those cases, software was greatly coupled with the hardware and large mainframes that possessed the required resources to run those software programs providing an interface through a terminal for the user to enter input and read the output.

As the hardware became more accessible to the mainstream public, less software was coupled with the hardware (mainly operating systems and hardware drivers) and the rest was left in many cases for the software application developer to write. Limitations still existed in which the software program was greatly tied to the operating system and, in some cases, the hardware it was implemented on. This, however, enabled the birth of server-client applications. Server-client application have a central hardware resource that handles the heavy computations and storage needs (the server) and connects to several user terminals that offer user-facing functionality such as the Graphical User Interface (GUI), authentication and permissions, local caching of data, etc. (the client).

With the widespread of the internet, more clients started to connect to larger and online servers (servers available publically and around-the-clock). This pushed the envelope for more requirements on the functionality provided through those applications for a much larger audience. Client-Server applications were purpose-built applications for targeted audiences that included hundreds of concurrent users on average. The demand for thousands of concurrent users could only be handled by introducing a lighter client and a more accessible server; this lead to web servers and internet browser-based applications. In the last twenty years, this architecture changed to accommodate for the increasing demands of user functionality, performance and availability. The first web applications were written on a web server that rendered HTML pages to internet browsers such as Microsoft Internet Explorer ® and Netscape ®. Later on, the internet browser started accommodating more functionality through technologies such as JavaScript, VBScript, ActiveX Controls and Java Applets. Through those technologies, browsers were able to handle more robust GUI functionality, better performing cacheable applications, and more secure connections to servers. This gave rise to what was known as Rich Clients or DHTML (D for dynamic). This approach was known as a 2-tier web architecture in which for any certain

application there were two nodes: the browser and the web server. As websites and web applications grew more popular in the commerce world, bottlenecks and performance issues started to arise by relying on single logical and physical source for computations (the server). A middle logical layer was introduced to separate the processing power from the data management and storage layer. The introduction of this middle layer gave rise to 3-tier web application architectures in which clients connect to the middle layer that aggregates and consume the required data and presents it to the requesting client.

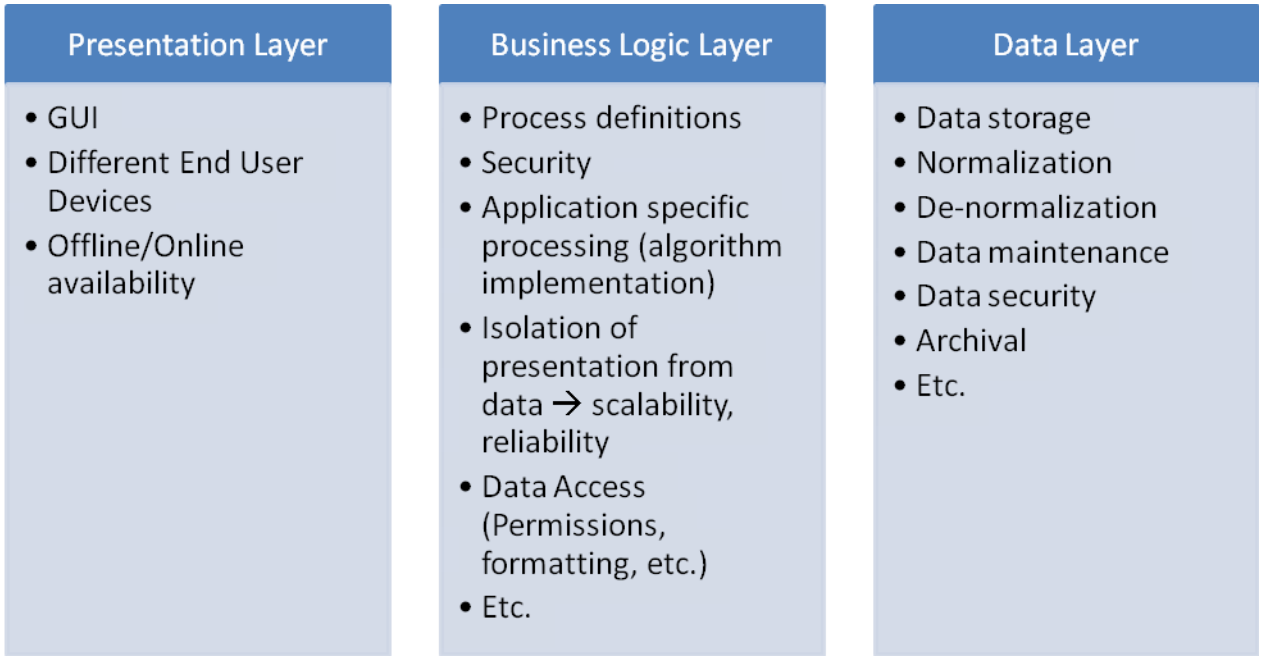


Figure 6. 3-Tier Application Architecture

This offered a more scalable architecture by which a physical implementation of the logical architecture could exist on several machines depending on where the load was expected. N-tier architectures which provide extra layers between the middle layer and the data layer for data security and encryption and extra layers between the middle layer and client for workflow and online-offline functionality presence were also introduced around the same time as 3-tier applications. 3-tier and n-tier applications also span the windows application software development not just web applications.

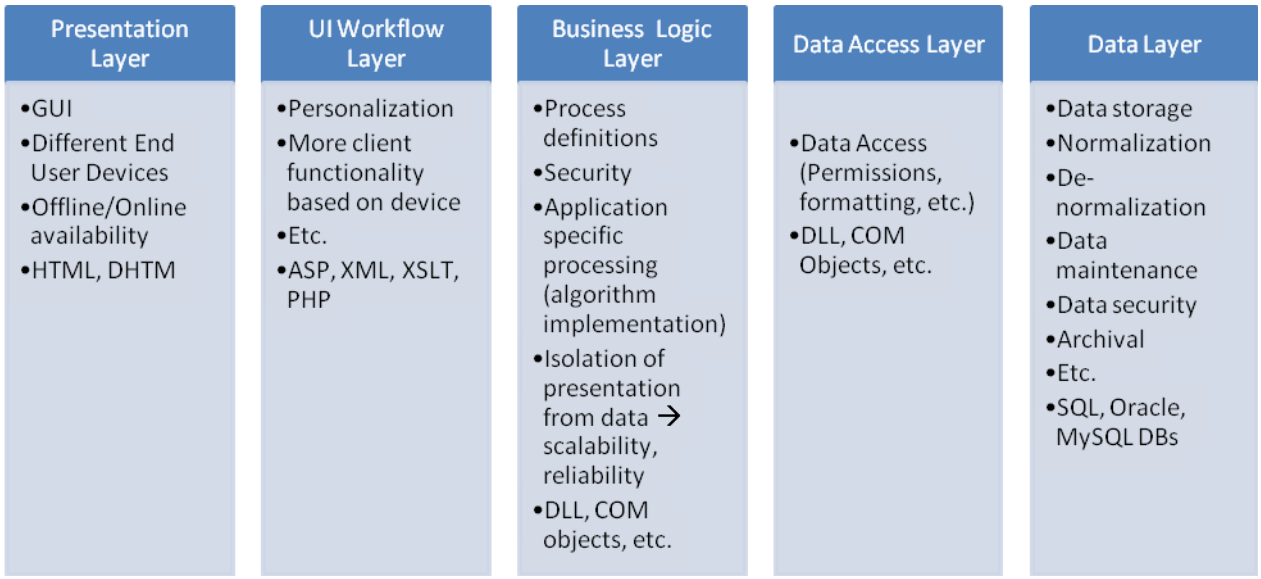


Figure 7. N-Tier Application Architecture

What made 3-tier and n-tier architectures possible was the use of object-oriented technologies such as Java and C++ that worked well with middleware technologies such as CORBA and COM+. The use of a middleware such as CORBA allowed for a Java object to be compiled and executed on a certain server or machine and have it reference objects that were compiled and executed on another machine (Raj 1998). Although those middleware technologies allowed for a distributed application to be implemented they did offer certain limitations that restricted the full decoupling between hardware and software. Java objects could only reference other Java objects. Objects in COM+ could only communicate if they were in the same network domain that controlled security and authentication. Objects could only communicate with specific object versions and upgrading certain objects in the distributed application required an upgrade of the whole application. The communication between the objects was controlled by the middleware technology and was not open to non-middleware or another middleware's components. Data elements were simply encapsulated as Java or C++ objects only accessible through the programming language that encapsulated them.

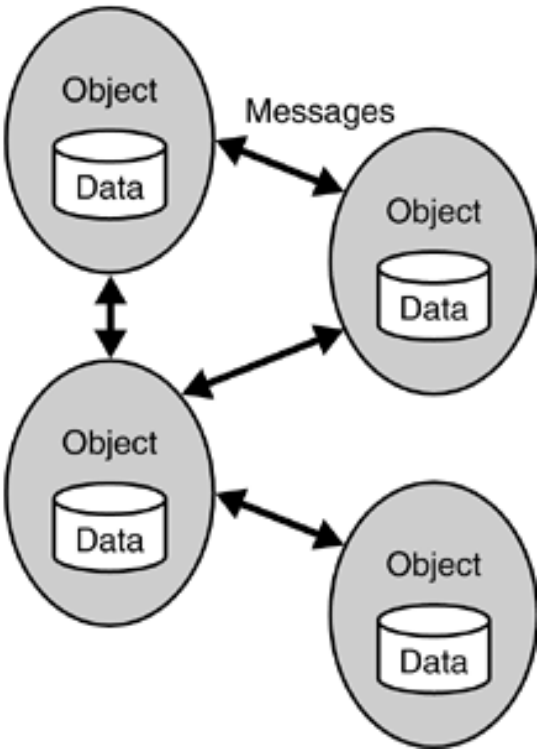


Figure 8. Object Oriented Technology

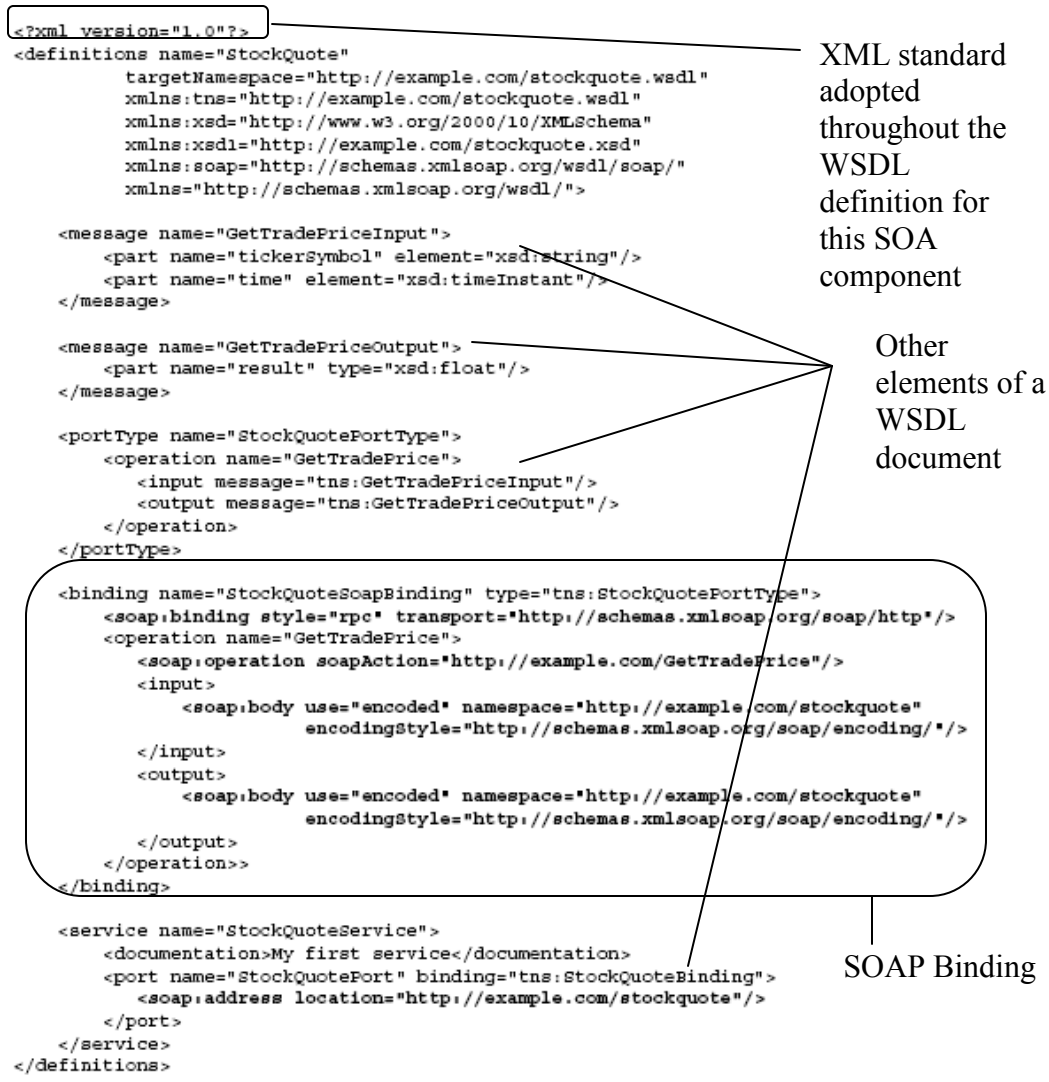
Figure 8 represents how object oriented programs written in a certain middleware such as CORBA would communicate. The interfaces between the objects are dictated by the CORBA middleware and the language in which the objects are represented (JAVA, C++). The need to incorporate other objects (from outside the JAVA/CORBA) realm was not possible; furthermore the capability of consuming objects developed by some other company or team was also not possible unless those objects were compile and executed under the same middleware in which they were to be consumed. The need to do those tasks was satisfied by introducing services or writing programs in a service oriented approach. In a service oriented program the middleware is not as restrictive. The middleware only works on transforming the code at run-time into a set of well established standards of communications. Those standards, known as SOAP, allow for a certain component to consume, as well as, provide its services to any other component that converses in SOAP. The two well known middleware technologies that provide this interfacing capability are Microsoft.Net and J2EE from Sun. Any component written to interface with either of those two middleware can interface with any other component written in either of those two middleware or other middleware that expose the SOAP interface. Several languages are supported on Microsoft.Net (J#, C#, VB.NET, C++, etc.) as well as Java is supported on J2EE; so language/middleware conformance is no longer an issue. Making use of this open architecture, services that handle security, authentication, as well as other administrative tasks are no longer part of the middleware but are services that can be consumed through the middleware. CORBA used to offer it own authentication feature, which all objects in CORBA had to use; with J2EE and Microsoft.NET there is no one particular authentication service; services can use any authentication service that satisfies their requirements and has a SOAP interface.

The SOAP Interface

A lot of the component oriented programming that relies on services relies heavily on the SOAP interface. In this section we present this briefly and direct the reader to the referenced material for a better understanding of SOAP.

Simple Object Access Protocol (SOAP) is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML-based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, what is standardized and most commonly implemented of SOAP has been in combination with HTTP and the HTTP Extension Framework. The encapsulation of service-based components adheres to SOAP standards and govern the creation of new service-based components. The SOAP used to define or encapsulate service is known as the Web Services Description Language (WSDL). WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate. WSDL standards are incorporated in the interface communications across service-based components. The abstraction provided by WSDL and the SOAP binding included in WSDL allow for the encapsulation of service-based components.

The example in Figure 10 is taken from the W3.org WSDL Standard documentation to illustrate how WSDL, SOAP and XML are used to produce a service component that queries the trade price of a stock ticker.




```
<?xml version="1.0"?>
<definitions name="StockQuote"
          targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

    <message name="GetTradePriceInput">
        <part name="tickerSymbol" element="xsd:string"/>
        <part name="time" element="xsd:timeInstant"/>
    </message>

    <message name="GetTradePriceOutput">
        <part name="result" type="xsd:float"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetTradePrice">
           <input message="tns:GetTradePriceInput"/>
           <output message="tns:GetTradePriceOutput"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetTradePrice">
           <soap:operation soapAction="http://example.com/GetTradePrice"/>
           <input>
               <soap:body use="encoded" namespace="http://example.com/stockquote"
                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
           </input>
           <output>
               <soap:body use="encoded" namespace="http://example.com/stockquote"
                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
           </output>
        </operation>>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
           <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>
</definitions>
```

Figure 9. SOAP-WSDL Representation

You will notice from this example that the interface is written in XML which is the standard means of communication between services. The document structure is governed by WSDL that contains the binding representation of SOAP to control the interface of this component with any other service-based component.

Service-oriented Middleware for cooperative vehicle-infrastructure systems

While several of the international efforts presented earlier in this report tried to interconnect vehicles and infrastructure through a pre-defined closed architecture comprising of a proprietary set of communication standards, well-defined application interfaces deeply tied to hardware specifications, and dedicated communication channels and bandwidth, we present in this section an alternate approach based on open architecture methods, loose coupling between software components and software and hardware components that leverages existing hardware and

communication media. This approach will be based on service-oriented programs that will prove capable of solving several of the concerns addressed in the section on Cooperative Vehicle-Infrastructure Systems.

Our proposed architecture for VII can be summarized in Figure 10.

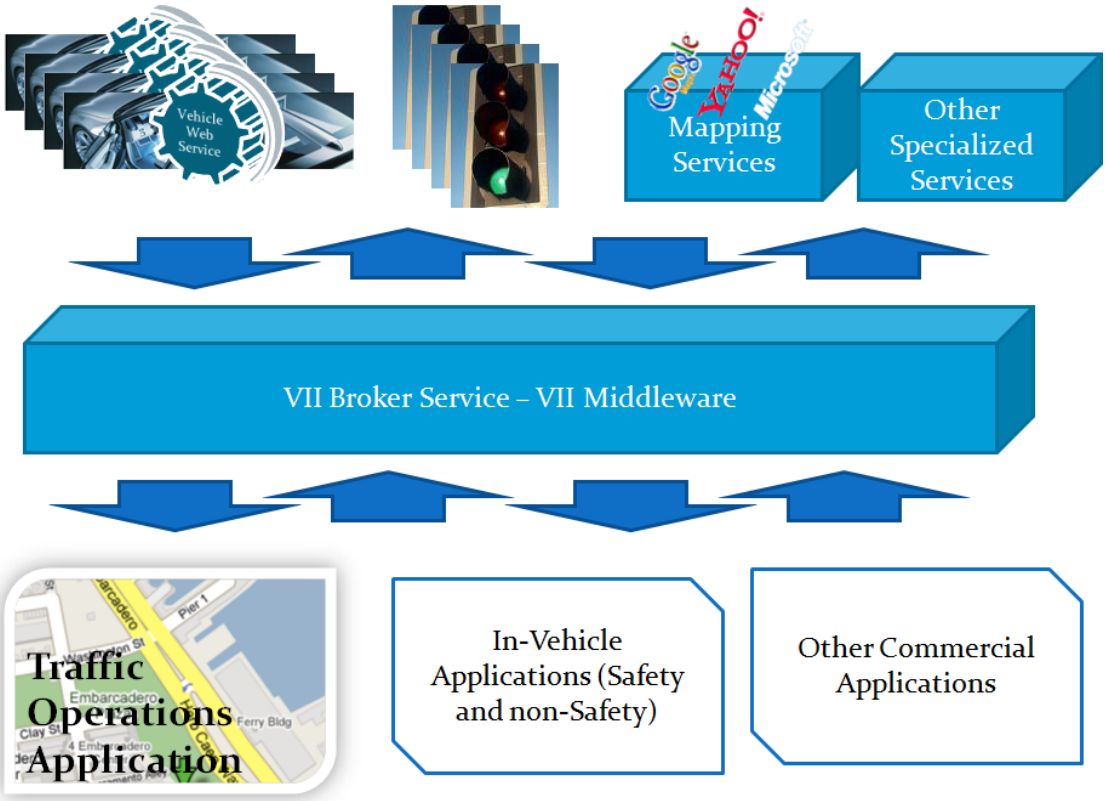


Figure 10. A Service-based Middleware for VII

In a service-based approach all our programs are written with a SOAP interface. We write services in vehicles, intersections as well as applications. We don't differentiate between an RSU and an OBU, the only difference between them is at the hardware level which for our services is just another service to interface with. The important logical component of the middleware is the VII Broker Service. The VII Broker is a service that allows for information exchange between the different services in the architecture. The VII Broker acts similar to a Domain Names Service (DNS) in the internet. Services requesting to communicate with other services, query the broker for the list of services and how to connect to them. In this relation we define two types of services: The consuming service is the service that queries the broker for other services; the consumed service is the service that is part of the query result set issued by the broker. From this we define two main functionalities for the VII Broker: one is to return the correct result set for the consuming service; two is to provide the WSDL files of the consumed services. The following example depicts how the consuming and consumed services work in relation with a VII Broker.

Take, for example, a Bay Bridge Speed Service, a service that averages the speed of vehicles on the Bay bridge in a certain direction (Eastbound or Westbound) and returns the result to the user (a human or another application). In this example the Bay Bridge Speed Service is the "consuming" service. It first queries the VII Broker for a list of vehicles on the westbound of the bay bridge. The VII Broker returns the result set of all vehicles that have registered with it with a GPS location on the Bay Bridge and a heading that is Westbound. The vehicles that satisfy the query are the "consumed" services. Each entry in the VII Broker query result set has a URL pointing to the vehicle's (or consumed service) WSDL file. The consuming service then uses the WSDL to get to the data from the vehicle.

Using the broker structure, all services (vehicles, road side units, intersections, etc.) have to first register with the broker to enable other services to consume them. The registration process requires each service to provide the current location of its WSDL file; this location is its URL and GPS location since the VII Broker queries are geo-spatial and relational queries.

Since we are using SOAP to communicate across the different services and SOAP relies on HTTP, TCP is the means by which the different services connect with each other. TCP can communicate over WiFi, Ethernet, GPRS and DSRC; this provides for a seamless communication layer that can change based on what is available for the VII network. We have also built a UDP-based communication layer that interfaces with the broker service thus allowing for a more flexible type of connection with less connection and overhead latencies than TCP. The applications that are built to communicate with the VII Broker and the services registered with it are seamless to what the underlying communication layer is; it is the broker that provides the best communication layer to the service requested.

In the following section we will present the prototype VII setup that was done at the Richmond Field Station – UC Berkeley to demonstrate the feasibility of a service oriented VII middleware.

Prototyping the Service-Oriented Middleware for Cooperative Vehicle-Infrastructure Systems

For our prototype we chose to implement the following scenarios:

- Capability to monitor vehicles on the road and an intersection from a vehicle on the road (A basic step in Intersection Assistance)
- Capability to monitor vehicles on the road and an intersection from an office (A Traffic Management Transaction)
- Capability to broadcast safety and non-safety related messages over DSRC to vehicles on the road (Simulcasting from the roadside for safety and non-safety applications)
- Capability to communicate between vehicles and road-side control elements (intersection) using DSRC and WiFi (Integration of protocols with DSRC)
- Capability to communicate between two vehicles using ad-hoc WiFi
- Capability to use different programming languages for different services

Implementation

For our prototype we used the following components:

- Two PATH vehicles each equipped with a USB GPS sensor connected to a laptop that had two WiFi cards and an Ethernet port.
- Two DSRC DENSO beta radios
- A 2070 controller connected to a live intersection and interfacing with a Linux machine over NTCP and using a sniffer to read the various phase parameters.
- An office desktop connected to the internet

The setup was as follows:

Each vehicle laptop was connected to the GPS sensor through the USB port and had the two WiFi cards configured to communicate over two different subnets. A windows service was installed on the laptops to read GPS data from the USB/Serial Port every 2sec and store the information in XML format in a local light-weight database server on the laptop. Another windows service was also installed on the laptop to detect the presence of a broker and if one is available (i.e. the vehicle laptop is online and has an IP address) to register the vehicle with the broker. The XML in Figure 11 is the message sent by the vehicle's registration service to the VII Broker's "RegisterMethod" method to register the vehicle.

```
"<Root><VehList><Vehicle><VIN></VIN><ProviderName>:</ProviderNa
me><ServiceURL>http://:/soa_vii_demo/NodeProperties.asmx?
wsdl</ServiceURL><Long></Long><Lat></Lat><Time></Time></Vehicle>
</VehList></Root>"
```

Figure 11. XML to register vehcile position and WSDL with VII Broker

The registration process would first query the local database for the most current GPS entry for that vehicle and would also query the IP stack for the laptop to figure out the current IP address and Port for the laptop's web service. The registration service would then call to the VII broker with the GPS and URL of the WSDL file on the vehicle laptop. The laptop was also setup with a local web service that ran on the laptop web server.

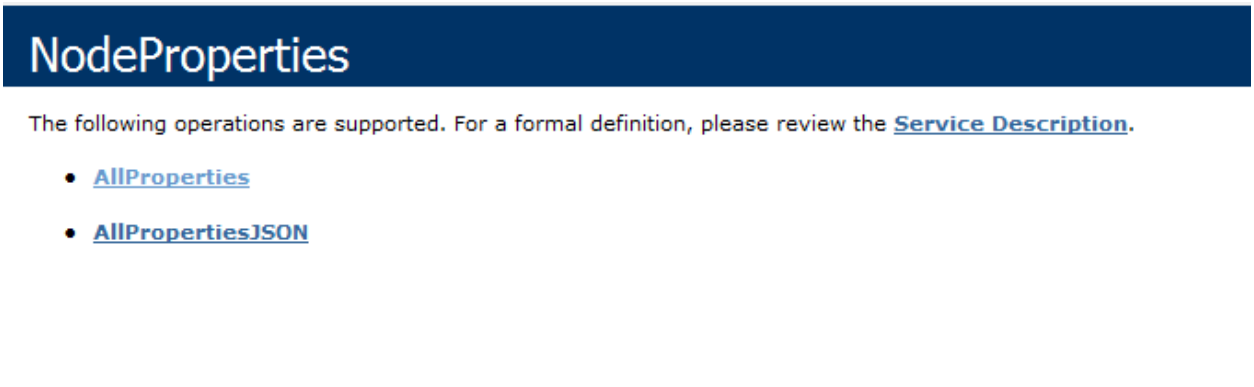


Figure 12. Vechcle Webserive available methods

The web service provides the user with information from the vehicle sensors' data; in our case speed and GPS location. The AllProperties method provides the information in XML, the AllPropertiesJSON method provides the same information in JSON which is another XML standard used for website JavaScript-based meshing allowing website developers to easily incorporate vehicle data into their web applications.

One of the vehicles was also equipped with the web service of a VII Broker which allowed other vehicles to register with it even if they are not in range with road-side units. This shows that the VII Broker-based architecture does not necessarily depend on a single broker but can communicate to a variety of brokers that could be in vehicles, road-side units, traffic control centers, etc. Borker-to-Broker communication protocols to ensure that brokers in close proximity had the same information was not implemented as part of this prototype; instead our services registered themselves with all brokers they could communicate with.

One of the vehicles was also connected to a DSRC radio through its Ethernet port allowing it to listen to DSRC broadcasted information on two different channels: one for safety applications and the other for commercial messages.

The intersection was equipped with a DSRC radio and a WiFi radio. The service written at the intersection broadcasted its information (cycle length and current phase information) through WiFi and DSRC over UDP.

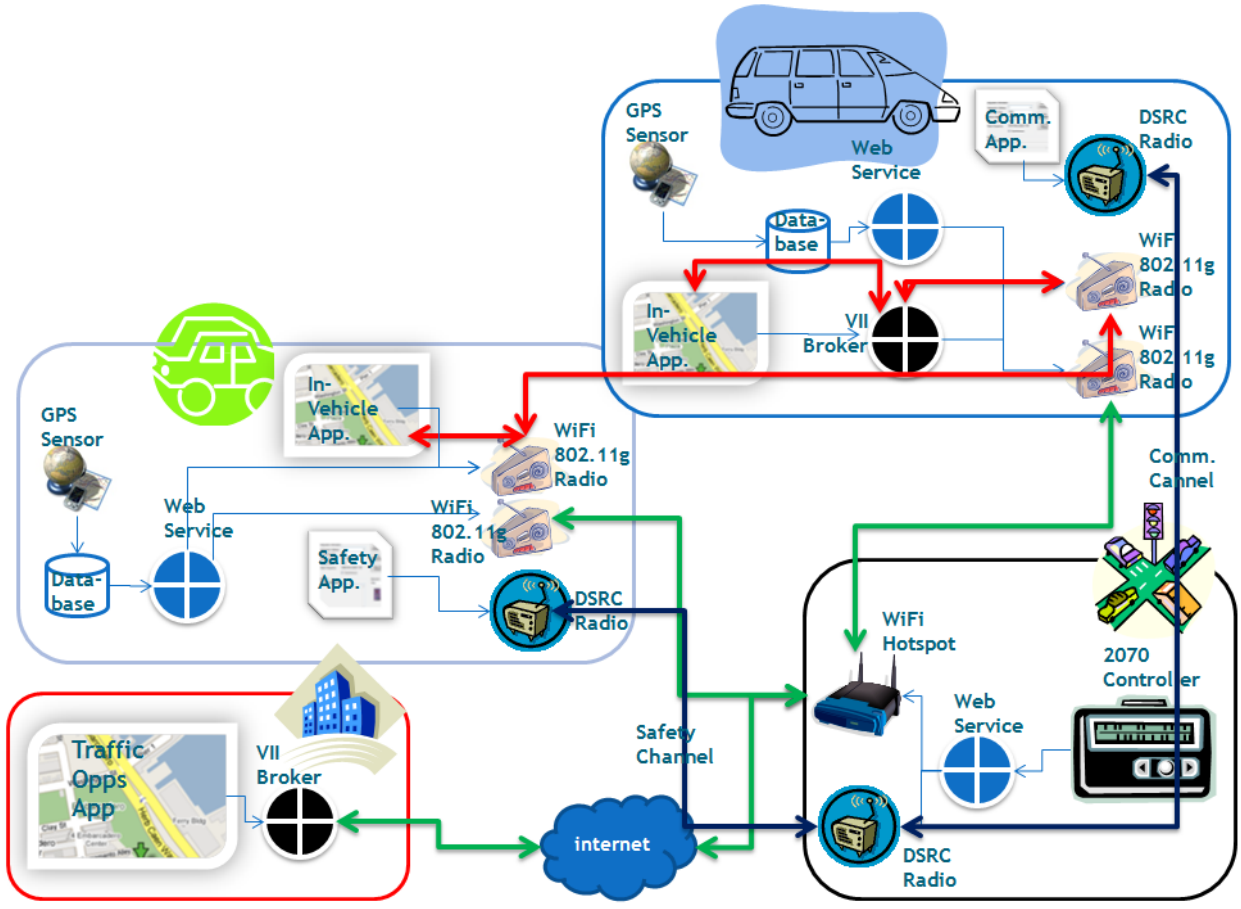


Figure 13. Service-based VII Middleware Prototype Setup

Figure 13 shows how the different components in the prototype communicated. Once a vehicle registration service detected a VII Broker in range (could be a VII Broker in another vehicle or a Vehicle Broker in an office) it would issue a RegisterMethod call; this would allow any querying application or service to connect to this vehicle. If the querying/consuming application requires the data from a certain set of vehicles it would query the VII Broker calling the

GetRegisteredVehicles method which would return all registered vehicles that that specific broker has. The application or service would then connect directly to those services one-by-one and consume their data. In this prototype, the map application inside the vehicle equipped with DSRC was able to communicate over WiFi to the adjacent vehicle and get its speed and location and communicate over DSRC with the intersection controller and gets its current cycle length and phase information. The same application was installed in another vehicle which did not have DSRC and that used WiFi to communicate with the other vehicle and the intersection. The choice of communication medium was left to the application in this case to choose what to use. Another application was installed in the vehicle that communicated only over UDP, this application utilized DSRC as the underlying layer for communication to read intersection phase information (safety) and some commercial (non-safety) data that was broadcasted by the intersection. This application was developed as a C# windows application and not as a service. This proves also that different types of applications can be written to the service-based middleware; choosing to write them as a service allows other applications to benefit from them, choosing to write them as close applications is also possible if the functionality that is being coded is to be kept hidden and unexposed to other services.

Conclusion

The implementation presented in this report has proved the main objectives that it was set out to deliver. First it reduces the complexities of the heterogeneous communication layer and data sources to a simple standardized *services* interface allowing the application developer to consume whatever data is generated from the sensors. In our implementation we choose web services as the middleware interface. Second, it solves sensor discovery problems by providing a geo-spatial query interface allowing the data consumer to locate the sensors of choice by providing a relational query engine. And third, it offers seamless connectivity between the application (data consumer) and the sensors (data source) by hiding the intricate details of the several communication standards from the application developer. In our implementation we utilize the concept of a message broker to develop the geo-spatial discovery and seamless connectivity features of the middleware. As a result, the application developer's task is reduced to just connecting to the middleware *services*. This simplification, however, comes at a certain cost incurred by the addition of the middleware layer.

In order to capture this overhead, we performed a modified setup of the components in the prototype with the middleware. The modifications aimed at accomplishing two main tasks: one, to isolate our measurements from any network overhead that might arise from network latency and congestion; second to increase the load on the broker to reach saturation stress levels allowing us to capture the overhead cost of a stressed broker.

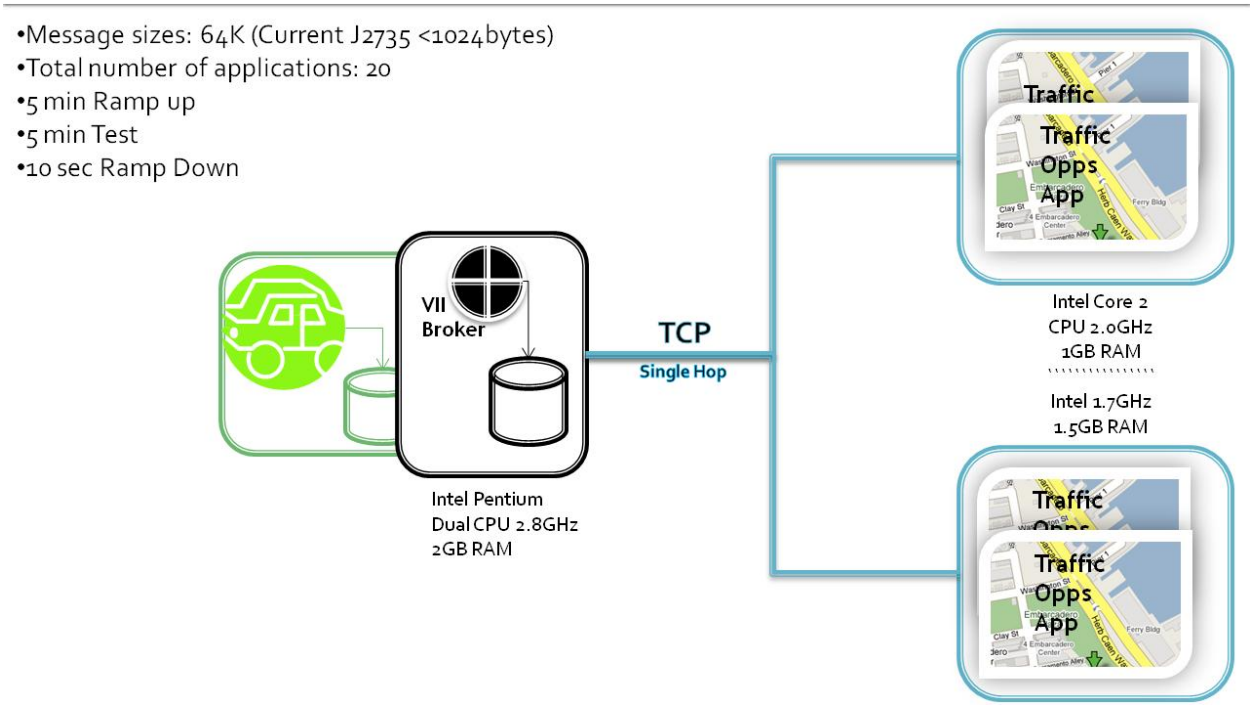


Figure 14. Setup used to measure middleware performance

The setup shown in Figure 14 was implemented in which a single vehicle sensor registered with the broker which existed on the same machine as the vehicle sensor. This eliminated any network overhead between the sensor and the broker. Two other machines were utilized as virtual application creators. Each machine simulated the existence of about 10 application running

simultaneously for a 5 min duration (the first 5 min. of the test were ignored for warming up the services, the second 5 min were used for performance measures and the last 10 sec were ignored for ramp down tasks). Given the setup in Figure 14 we were able to arrive at the following numbers:

- Number of transactions in 5min: 71,018
- Rate of transactions per second: 236.727
- Average response time per transaction: **~122 msec**
    - Time spent over TCP and wire: 1.3-2.6 msec
    - Time spent serializing/deserializing SOAP: 47.143* msec
    - Time spent processing request: 72.657 msec

*Tests done by Microsoft and Sun on SOAP average 50msec

From the above measures, the 122 msec average response time can be considered as the cost of using the middleware. When considering applications such as probe vehicles and real-time traffic control systems such as RHODES (Mirchandani 2001) and OPAC (Gartner 1983) the minimum transaction update rate is in the order of 20-30 sec which makes our middleware with range for such applications.

References

CVIS. *CVISProject.org.* 2007. http://www.cvisproject.org (accessed September 5, 2007).

Gartner, N.H. OPAC: A Demand-Responsive Strategy for Traffic Signal Control. Transportation Research Record 906. TRB, pp. 75-81. 1983.

Estrin, D., R. Govindan, J. Heidemann, and S. Kumar. "Next Century Challenges: Scalable Coordination in Sensor Networks." *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM '99).* Seattle, Washington, August 1999.

ISO. *ISO TC204 WG16 CALM - The CALM Scope.* 2007. http://www.calm.hu/ (accessed September 5, 2007).

ITS Joint Program Office. *VII Architecture and Functional Requirements version 1.1.* Washington, D.C.: FHWA - US DOT, 2005.

ITS U.S. DOT. *VII Concept of Operations - ITS.* November 2006. http://www.its.dot.gov/vii/vii_concept.htm (accessed June 11, 2007).

Mietzner, Rudolf. "CVIS - Cooperative vehicle-infrastructure systems." *COM Safety: Newsletter for European ITS Related Research Projects*, July 2007: 3,5.

Mirchandani, P.B. andHead, K.L. , "A Real-Time Traffic Signal Control System: Architecture, Algorithms, and Analysis," *Transportation Research Part C,* vol. 9, no. 6, 2001, pp. 415–432.

Raj, Gopalan Suresh. *A Detailed Comparison of CORBA, DCOM and Java/RMI.* 1998. http://my.execpc.com/~gopalan/misc/compare.html (accessed September 5, 2007).

Reding, V. "Speech delivered at the Intelligent Car Launching Event." *The Intelligent Car Initiative: raising awareness of ICT for Smarter, Safer and Cleaner vehicle.* Brussels, 2006.

Setsuo, Hirai. "Smartway project towards the next generation ITS." *COM Safety: Newsletter for European ITS Related Research Projects*, July 2007: 3.

The CAMP Vehicle Safety Communications Consortium. *Vehicle Safety Communication Project Task 3 Final Report: Identify Intelligent Vehicle Safety Applications Enabled by DSRC.* Washington, D.C.: National Highway Traffic Safety Administration - US DOT, 2005.

Tindale-Biscoe, Sandy. "RM-ODP Enterprise Language (ISO/IEC 15414 | | ITU-T X.911)." *ITU-T/SG17 Meeting.* Geneva, 2002.

US Department of Transportation. *Vehicle-Infrastructure Integration (VII).* 2007. http://www.its.dot.gov/vii/vii_overview.htm (accessed Feb. 19, 2007).