A Flexible System Architecture for Acquisition and Storage of Naturalistic Driving Data

Asher Bender, James R. Ward, Stewart Worrall, Marcelo L. Moreyra, *Member, IEEE*, Santiago Gerling Konrad, Favio Masson, and Eduardo M. Nebot

***Abstract*—Innovation in intelligent transportation systems relies on analysis of high-quality data. In this paper, we describe the design principles behind our data management infrastructure. The principles we adopt place an emphasis on flexibility and maintainability. This is achieved by breaking up code into a modular design that can be run on many independent processes. Message passing over a publish–subscribe network enables interprocess communication and promotes data-driven execution. By following these principles, rapid prototyping and experimentation with new sensing modalities and algorithms are possible. The communication library underpinning our proposed architecture is compared against several popular communication libraries. Features designed into the system make it decentralized, robust to failure, and amenable to scaling across multiple machines with minimal configuration. Code written using the proposed architecture is compact, transparent, and easy to maintain. Experimentation shows that our proposed architecture offers a high performance when compared against alternative communication libraries.**

***Index Terms*—Intelligent transportation systems, naturalistic driving data, data acquisition, data storage, visualization, communication, LCM, ZMQ, RabbitMQ, ROS.**

I. Introduction

RESEARCH fuels innovations in intelligent transportation systems (ITS). Careful observation, design and testing enable this research to progress. For this progression to be productive, engineers and researchers need access to high quality data. As a result, reliable and flexible methods for data acquisition, storage and analysis are key to progressing innovation.

Maintaining data infrastructure is difficult and time consuming. As data systems grow in scale and complexity, managing the system becomes increasingly resource intensive and prone to failure. Developing algorithms or software to handle reading, processing and storage of new sensing modalities can become an onerous and risky task. Some algorithms or sensing modalities may prove to be useful and will become integrated into the system. Others may only provide a small return on the effort invested in experimentation. In the worst case, an algorithm or sensor will provide no useful information and will be removed from the system altogether.

The primary focus of this paper is data acquisition. The data management infrastructure proposed in this paper has been designed to facilitate research. In a research environment, frequent experimentation with new sensors and algorithms is common. Reducing the burden of software and hardware integration in systems under constant development allows engineers and scientists to focus their energy on research and rapid prototyping. This consideration places an emphasis on designs that are flexible, maintainable and reliable.

The main contribution of this paper is a strategy for designing software and hardware for collecting naturalistic driving data. An overview of the design of our data management infrastructure is shown in Fig. 1. The system architecture is split into two broad categories: in-vehicle data acquisition and off-vehicle data acquisition. The in-vehicle data acquisition system is illustrated on the left side of Fig. 1. Data collected from vehicle sensors is automatically logged by an automotive computer. Any process running on the automotive computer can access or produce data broadcasts. The off-vehicle data acquisition system is illustrated on the right side of Fig. 1. External infrastructure is able to opportunistically harvest data from vehicles and insert the records into a centralized database for later visualization and analysis. Combined, the in-vehicle and off-vehicle systems are an effective method for accumulating big data sets due to their lack of reliance on human operators.

The remainder of the paper is structured as follows. Section II discusses related work in ITS data acquisition systems. In Section III we propose a generic software architecture for acquiring data in ITS applications. In Sections IV–VI our in-vehicle and off-vehicle data collection systems are described. In Section VII we briefly describe how data collected from these systems can be visualized. Our system is evaluated in Section VIII. Finally, Section IX presents our conclusions.

II. Related Work

Previous research projects have developed sophisticated mechanisms for data acquisition, storage, analysis and visualization. The capacity and flexibility of these systems has allowed researchers to push the boundaries of ITS and perform state-of-the-art research. Notable examples include Bertha Benz [1], a car capable of operating autonomously in city traffic

Manuscript received October 16, 2014; revised April 22, 2015, July 17, 2015, and November 23, 2015; accepted January 19, 2016. Date of publication February 24, 2016; date of current version May 26, 2016. This work was supported by the Australian Research Council Linkage Project under Grant LP120100700. The Associate Editor for this paper was W.-H. Lin.

A. Bender, J. R. Ward, S. Worrall, and E. M. Nebot are with the Intelligent Vehicles and Safety Systems Group, Australian Centre for Field Robotics, The University of Sydney, Sydney, N.S.W. 2006, Australia (e-mail: a.bender@acfr.usyd.edu.au).

M. L. Moreyra is with Universidad Nacional del Comahue (UNCo), Q8300IBX Neuquén, Argentina.

S. Gerling Konrad and F. Masson are with the Departamento de Ingeniería Eléctrica y Computadoras, Universidad Nacional del Sur, B8000CPB Bahía Blanca, Argentina.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TITS.2016.2524523

1524-9050 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

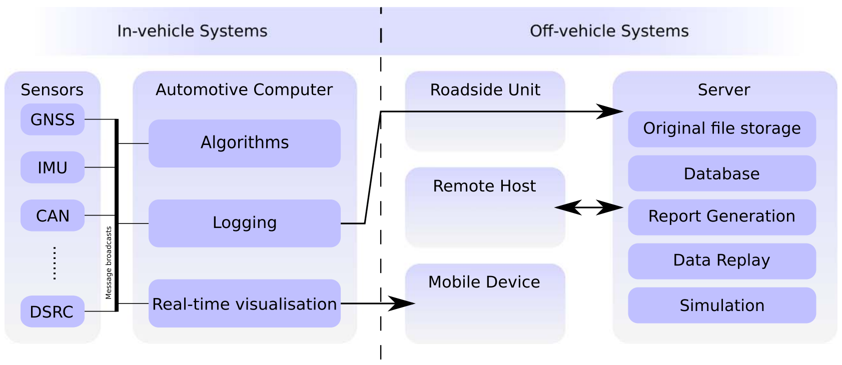


Fig. 1. Overview of data management architectures.

and the competitors in the DARPA Urban [2] and Grand [3] Challenge.

In addition to single vehicle systems, there have also been a number of large-scale vehicle-to-vehicle (V2V) trials. Rather than focusing on autonomy, these trials investigated the collection of naturalistic driving data [4] and evaluated V2V technologies [5]. Analysis of big data from vehicular fleets has also been performed to extract patterns in trip behavior, temporal relationships and trip distributions from trajectory data gleaned from taxis [6], [7].

Although previous research projects overcome the challenges of maintaining data infrastructure, the strategies behind their success are often over-looked. Publications documenting these platforms typically provide a high-level overview of the system design—including topics such as sensing payloads, perception routines, mission planning strategies and route management. Little detail is provided on the architectures that provide logging, processing, analysis and data visualization.

For research groups developing new systems, insight into the design of existing systems will help ensure successful data collection, storage, analysis and visualization strategies are adopted. To the authors' knowledge, there have been no papers dedicated to this important aspect of experimental work in the ITS field. This publication addresses this gap in the literature and documents the architecture behind the core functionality of our data management system—acquisition, storage, analysis and visualization. In discussing the design and philosophy behind our approach we hope to allow others to benefit from the lessons learned through our experiences.

III. Software Architecture

Developing software for managing concurrent and complex tasks is time consuming and difficult. To ensure the success and longevity of a system, the software must be extensible, easy to maintain and reliable. In this section we propose a software architecture for reading, processing and storing data in ITS applications.

The proposed software architecture is designed to place an emphasis on flexibility, maintainability and extensibility. The ability to add new sensors and algorithms to the system without requiring large changes to the code-base makes the system an effective tool for research and development. Two key concepts underpin the design philosophy of the software: the publish–subscribe paradigm and inter-process communication. Section III-A motivates the publish–subscribe paradigm as a method for writing loosely-coupled, maintainable code. Section III-B shows how inter-process communication can be used to make the system efficient and fault tolerant.

A. Publish–Subscribe

By breaking up software into units of functionality, small portions of code can be independently developed and maintained. These units can be loosely coupled by defining how information is transferred from one unit to another. We pursue loose coupling by using the publish–subscribe paradigm. The publish–subscribe paradigm is a design pattern for transferring data from one object to another. Objects, known as *publishers*, can make data available by issuing 'publish' events. Other objects, known as *subscribers*, can receive data by subscribing a callback function to publish events.

For the publish–subscribe paradigm to operate effectively, subscribers must know how to interpret the data they receive via callbacks. This is done by defining a format for each data type used by the system. These formats are called *messages*. An object that generates publish events may only pass one message type to its subscribers. Following this restriction, by registering a callback with a publisher, the subscriber acknowledges that it will receive a particular message. Although publish events are limited to one message type, an object may host an arbitrary number of different publish events. Messages passed around the system inherit from a base message class. Inheritance ensures all messages are formatted correctly and contain common attributes. Defining new message types becomes a trivial task of inheriting from the base message class and specifying what fields of data the message will contain.

The advantage of following the publish–subscribe paradigm is that publishers do not need to consider how the data will be manipulated once it has been published. Similarly, subscribers do not need to consider how the data is generated before it is published. Apart from promoting loosely coupled and reusable objects, the publish–subscribe paradigm also allows for event driven software. Software written to read sensor information can make the data available and drive activity within the system by issuing publish events. Callbacks designed to process the sensor data will only be executed once the data has been made available via the publish events.

B. Inter-Process Communication

The publish–subscribe paradigm described in Section III-A allows a collection of loosely coupled objects to share data within a single process. In a complex system, handling all functionality within a single process is undesirable as it creates a single point of failure that is difficult to diagnose. By partitioning the system into well defined asynchronous modules, the system can become distributed.

Since processes operate in independent memory spaces, the publish–subscribe mechanism (Section III-A) cannot be applied directly. Inter-process communication must be used to extend the publish–subscribe paradigm from the object-level to the process-level. To transfer data, the processes need to share a common communication mechanism. Any communication library which supports many-to-many data distribution such as multicast [8] or star topologies [9], [10] can be used. In our proposed framework, IPv6 multicasts are used. This transport mechanism is discussed in greater detail in Section VI.

Using inter-process communication to link processes together decentralizes the functionality of the system. The processes can run on heterogeneous hosts, hardware architectures, operating systems and languages. Distributing the system across multiple processes also allows the software to take advantage of the ever increasing number of processing cores available for concurrent execution of code. In addition to being efficient, the multi-process strategy also enhances the integrity of the system. Since each process has been isolated and allocated its own resources, if one process fails, it will not cause the whole system to crash. Similarly, a resource hungry process will not block or prevent other processes from accessing a fair share of the system resources.

The topology of the communication network is structured such that each message type is associated with a specific uniform resource identifier (URI). Again, this one-to-one restriction makes the message passing strategy explicit. By listening for broadcasts on a particular URI, the listener has acknowledged that it will receive a particular message type. Although a one-to-one mapping exists between URI and message type, a many-to-many relationship can exist between broadcasters and listeners. Multiple processes can broadcast to a single URI while many processes listen for broadcasts. The ability to subscribe and publish data to a small set of known URIs makes creating a many-to-many relationship between broadcasters and listeners both simple and maintainable. Simplifying the ability to create complex networks of data traffic promotes the development of systems capable of collecting big data.

In a system containing multiple message types, processes will need to identify what messages are available and where to locate them. The network topology is made available to the system via a message specification which maps from message type to URI. Processes wishing to participate in and comply with network traffic must adhere to the specification. To disambiguate where messages are generated in a many-to-many topology, broadcasters and listeners can use *topics* during transmission to filter messages. For example, in a vehicle equipped with forward and rear facing cameras, the image messages could be disambiguated by associating broadcasts with the topics “forward” and “rear” respectively.

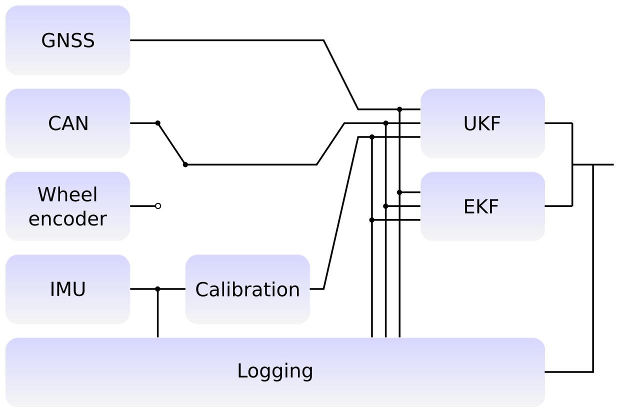


Fig. 2. Example system using inter-process communication to produce state estimates. Each block represents an independent process capable of producing and/or consuming data. The lines connecting the processes represent message broadcasts. The connections formed between processes illustrate dependencies within the system. To illustrate the flexibility of this architecture, this example shows a UKF and EKF running in parallel with a switchable source of data for speed information. The bottom block represents a logging process recording all messages in the system.

The inter-process communication strategy is demonstrated in Fig. 2. Each block in the diagram represents an independent process. The lines connecting the processes represent message broadcasts. As the diagram shows, processes can be added or removed from the system. For instance, vehicle speed can be provided by either the controller area network (CAN) bus or encoders on the wheels. The switch from one source of data to another is transparent to the rest of the system. Similarly, algorithms can be added to or removed from the system transparently. On the far right of Fig. 2, state estimates are produced in parallel using an unscented Kalman filter (UKF) and an extended Kalman filter (EKF). The UKF and EKF estimates can be disambiguated using topics. Normally only a single position filter that has been tuned to the particular vehicle model would be run.

IV. In-Vehicle Data Acquisition

In-vehicle data acquisition is performed using the software architecture described in Section III. The type of data available for logging is discussed briefly in Section IV-A. The automotive computer used to run our software architecture and log sensor data is discussed in Section IV-B. Implementation details of the logging system are discussed in Section IV-C.

A. Sensors

Modern advanced driver assistance systems (ADAS) are monitoring increasingly sophisticated driving tasks and providing support and functionality such as lane departure warnings, adaptive cruise control and automatic parking. Next generation ADAS such as detecting pedestrians [11], inferring driver intent [12], [13] and advanced collision avoidance systems [14], [15] require complex sensing modalities to gather the data they require to fulfill their objectives.

Developing next generation ADAS will require experimenting with multiple, possibly novel, sensing modalities. One of

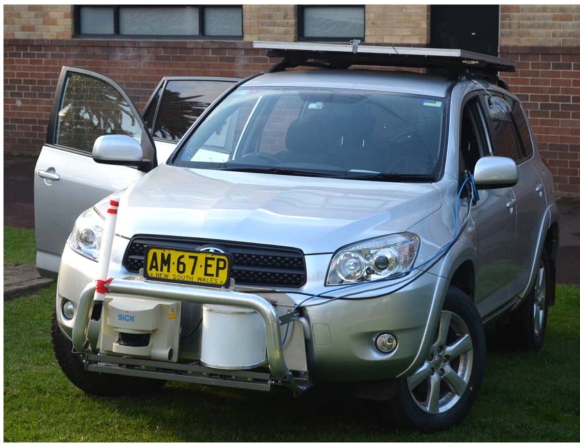


Fig. 3. Research vehicle equipped with multiple sensing modalities, including LIDAR, RADAR, GNSS and an inertial system (3-axis gyroscopes and accelerometers).

our research vehicles is shown in Fig. 3 as an example of an experimental system with many sensing modalities. The sensors itemized in the following list are commonly found in ITS research programs.

- Inertial measurement unit (IMU)
- Global navigation satellite systems (GNSS)
- LIDAR
- RADAR
- MobileEye
- CAN messages
- Dedicated short-range communications (DSRC) radio module

Aggregating multiple sources of data commonly found in ITS research platforms motivates the need for a system designed to centralize and record all of the sensor data generated by a vehicle. The automotive computer described in Section IV-B fulfils this need and acts a flexible central node for processing sensor data.

B. Automotive Computer

Sensor data in our research vehicles are logged and processed on a custom assembled automotive computer. The hardware components of this system are shown in Fig. 4. For the sake of brevity, we refer to this hardware as the *Blackbox*.

The *Blackbox* is based on a mini-ITX, x86 motherboard in a ruggedized computer housing. The hardware was selected to fulfill the design criteria of logging sensor data, light processing and network provision. Consumer grade components are used internally to reduce the cost, lead times and simplify upgrades in response to increased computing requirements. Currently the *Blackboxes* are equipped with an Intel G3420 processor and 4GB of RAM. Data is logged to a solid-state hard disk drive.

Network connectivity is provided by a gigabit switch and a WiFi USB dongle. V2V and vehicle-to-infrastructure (V2I) communication is provided by a Cohda Mk4 DSRC radio module. The network capability provided by the *Blackbox* allow it to act either as a central node for coordinating network traffic or part of a networked system of computers.

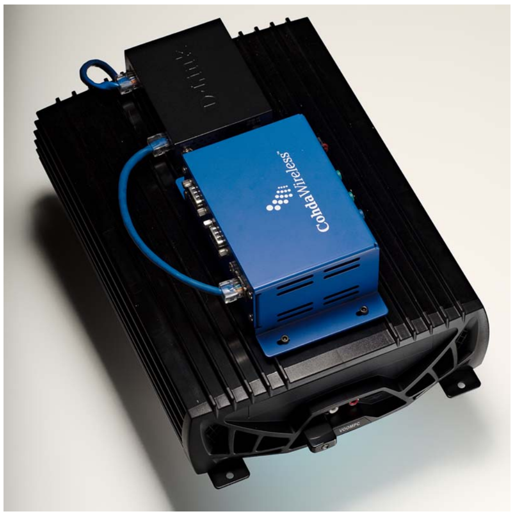


Fig. 4. Automotive computer including a gigabit switch, USB WiFi dongle and DSRC radio module. This hardware forms an easily deployed system for logging naturalistic driving data.

C. Data Logging

Data analysis is essential for understanding the characteristics of natural driving scenarios and developing new algorithms. As a result, collecting quality data is fundamental to facilitating research and advances in ITS applications. One of the primary objectives of the *Blackbox* is to provide a mechanism for logging data. This is done using the flexible software architecture described in Section III.

As described in Section III-B, it is possible for processes to identify what messages are available in the system and where to locate them. This makes it possible for any process to access the network traffic of the entire system. Since there is a one-to-one mapping between message type and URI, all of the network traffic can be logged in real-time by subscribing a listener to each URI in the system. When a message is received the listener records the time that has elapsed since logging started, the broadcast topic and message payload to a plain text file. This strategy results in a log file for each message type broadcast within the system. A process logging the network traffic is shown as the bottom block in Fig. 2.

High frequency or large capacity data such as inertial measurements and images can cause the log files to grow in size rapidly. Large data files are not conducive to opportunistic data harvesting. Vehicles may only be in range of a roadside unit (RSU) for a few seconds making it impossible to transfer large log files. To make the log files amenable to opportunistic harvesting, the log files can be split by the number of data entries, by time or by both. The infrastructure for opportunistic data harvesting is described in Section V-A.

To prevent uninteresting data from being recorded, the logging system is only initiated when the engine is running. Similarly the logging system is terminated when the engine is turned off. While promoting efficient use of resources, this automatic logging system is also convenient as it does not depend on a human operator. Datasets are defined by engine-on and engine-off events. Each dataset is stored in a newly created, time stamped directory.

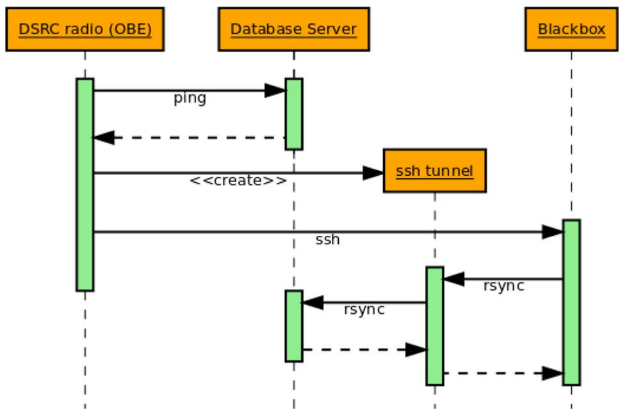


Fig. 5. Sequence diagram of opportunistic data harvesting. Events occur in time sequence down the diagram.

V. Off-Vehicle Data Acquisition

The in-vehicle data acquisition system described in Section IV is capable of automatically recording a complex payload of sensor data. While these systems can be used in vehicle trials, their usefulness is hampered by the need for human operators to manually retrieve data files recorded by the system on a regular basis. Manual intervention is not amenable to accruing big data.

This section describes additional infrastructure, outside the vehicle, that transforms the system into one that is capable of automatically harvesting data from the in-vehicle data acquisition systems. Again, the software architecture described in Section III is used as a backbone for the system. Section V-A describes how data is automatically harvested from in-vehicle data acquisition systems. Section V-B describes how the harvested data is stored in a central database.

A. Opportunistic Data Harvesting

An important feature of our system is that off-vehicle data acquisition systems are able to harvest data logged by in-vehicle data acquisition systems. Our off-vehicle data acquisition systems are implemented in RSUs equipped with DSRC and connected to a centralized database. Once a vehicle is able to connect to a RSU using DSRC, data can be transferred to a centralized database. We have demonstrated the value of such an approach in the context of mining and mining safety [16], [17] and apply a similar approach to ITS.

A sequence diagram of the process is shown in Fig. 5. In order to avoid creating and tearing down routes on the automotive computer whenever the DSRC radio establishes communication with an RSU, the sequence is initiated and controlled by the DSRC radio. The first step is to determine whether communications can be established with the database server via the RSU. The DSRC unit periodically pings the database server and listens for a response. If a response is received it begins the data harvesting process. This is done by establishing an *ssh* tunnel between the automotive computer and the database server. This design was chosen because the DSRC radio already needs to connect to the automotive computer via *ssh* to initiate file transfers, so the tunnel setup can be integrated into this process. Once the tunnel is established, the automotive computer runs *rsync* to transfer files. Files that are successfully transferred are deleted from the automotive computer. This eliminates the need for periodic maintenance to ensure there is appropriate disk space on the automotive computer. The database server periodically polls the directory where uploaded log files are stored and any new ones are processed using the method described in Section V-B.

Since the in-vehicle data acquisition system logs different message types in separate files, the harvesting system can be configured to prioritize certain messages types. This is done so that information can be recovered from vehicles while respecting bandwidth limitations of the V2I network. Message priorities can be configured for the requirements of the particular deployment. For example in a real-time monitoring application, state information such as position and speed could be prioritized over bandwidth intensive camera data.

A natural result of this architecture is that as the number of RSUs is increased, the latency between when a vehicle logs data and when it is available in the database decreases. If infrastructure is deployed such that a given vehicle is always in contact with an RSU then the data from the vehicle will be available in real-time, without any change to the fundamental underlying configuration of the system. This opens up the possibility of traffic monitoring and management in real-time.

B. Database

The database server has a PostgreSQL/PostGIS database. As log files are harvested from the vehicles they are parsed and stored in the database. The log files are also stored on the server for backup purposes. The geographic information system (GIS) extensions in the database make selecting data on a geospatial basis far simpler than searching through the log files.

Each message in the raw data is processed in a straightforward manner. If the message contains geospatial information, it is extracted from the message and combined into a single PostGIS geometry entity. Messages in this category include global navigation satellite systems (GNSS) messages, received DSRC Basic Safety Messages (BSM) and messages from position filters such as UKFs. Since PostgreSQL can handle JSON as a native type and messages are stored as associative arrays, non-geospatial data are translated into a JSON representation and injected directly into the database. The advantage of such an approach is that messages with optional fields can be stored, indexed and queried in the database. This provides the combined benefits of being able to store semi-structured data and the power of SQL indexing and geospatial queries. It is this flexibility that allows the reporting and visualization discussed in Section VII.

The way data is transferred from vehicles to the database allows sharding to be used for load balancing and scalability. For example, many database instances can be established at different sites with each database connected to local RSUs. Load balancing on the individual databases is achieved by connecting only a small number of RSUs that have a large amount of vehicular traffic (and therefore large data downloads). These database shards can then agglomerate their data into a single instance using established database sharding tools. The scalability of the system is thus only constrained by the database backend, and not the V2I network itself. Advanced techniques for the storage, replication and load balancing of big data from vehicular systems is discussed in [18] and could be applied to our architecture.

VI. Communication

The software architecture discussed in Section III and the hardware discussed in Sections IV and V require well defined communications protocols. In Section VI-A we nominate IPv6 UDP multicasting for use as a network transport mechanism. In Section VI-B we discuss how the network can be configured to facilitate IPv6 UDP multicasting between devices.

A. Software

The inter-process communications architecture described in Section III-B requires a network transport that can support a many-to-many data distribution. The transport mechanism we have adopted for inter-process communication is IPv6 UDP multicasting. This is a similar design to the lightweight communications and marshalling (LCM) [8] library which uses IPv4 UDP multicasting. IPv6 is preferred because the creation of multicast groups and routes is greatly simplified over IPv4.

Within the software architecture, each message type is associated with a unique multicast group. Publishers and subscribers can begin to transmit or receive messages by subscribing to a multicast group. Using multicast groups in this way is advantageous as the decision to forward a message across the network is transparent to the high-level software architecture. These decisions are handled at the network level by switches and routers in the network.

Since our transport mechanism relies on UDP rather than TCP connections, there is no guarantee of message delivery or message ordering. UDP is also unable to transmit large messages, such as big images. To handle large payloads, a splitting and marshalling system has been implemented. Again, this approach is similar to LCM.

B. Configuration

In order to make the network as transparent to the software architecture as possible, an addressing scheme was chosen that makes configuration and service discovery easy. The addressing scheme for a typical in-vehicle data acquisition system and RSU node is shown in Fig. 6 and summarized in Table I.

Each in-vehicle data acquisition system has three main components, as can be seen in Fig. 4 and the left-hand side of Fig. 6. These are the Ethernet switch, automotive computer, and DSRC radio module. Ethernet connections are assigned addresses in the `2001:db8:0:0nnn::/64` subnet—where *nnn* is the equipment unit number. The automotive computer is assigned the static IP address `2001:db8:0:0nnn::1/64`, and the DSRC radio `2001:db8:0:0nnn::2/64`.

Any devices connected to the network via the Ethernet switch can auto-configure their IP address based on the prefix advertised by a *radvd* daemon running on the automotive computer. The *radvd* daemon also advertises the `2001:db8:0:1nnn::/64` prefix on the WiFi interface. A route to the `2001:db8:0:0nnn::/64` and `2001:db8:0::/64` subnets is provided allowing WiFi connected devices to talk directly to the DSRC radio. The DSRC radio is able to provide V2V and V2I functionality by establishing ad-hoc connections with other DSRC nodes. The DSRC radio behaves like a standard network interface and is allocated the address `2001:db8:0:2nnn::2/64`.

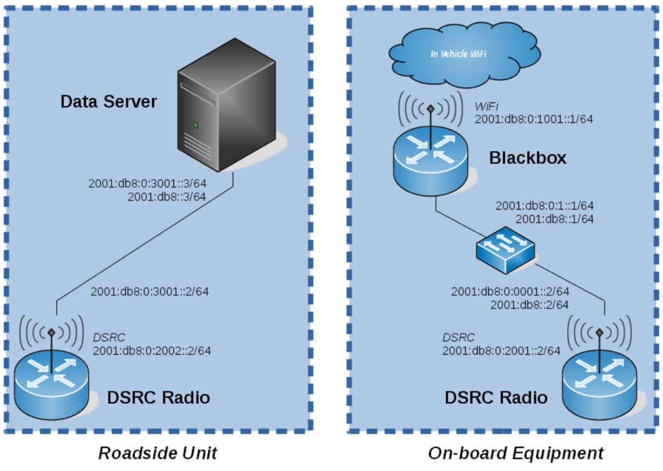


Fig. 6. IPv6 addressing scheme. Addresses are represented in the 2001:db8::/32 documentation namespace. Actual implementation of this scheme can be in any prefix.

TABLE I
Summary of IPv6 Addressing Scheme

Device	Interface	Address Type	Address
Blackbox	Ethernet	unique	2001:db8:0:0nnn::1/64
		anycast	2001:db8:0::1/64
	WiFi	unique	2001:db8:0:1nnn::1/64
DSRC radio	Ethernet	unique	2001:db8:0:0nnn::2/64
		anycast	2001:db8:0::2/64
	DSRC	unique	2001:db8:0:2nnn::2/64
Database server	Ethernet	unique	2001:db8:0:3nnn::3/64
		anycast	2001:db8:0:3000::3/64
			2001:db8:0::3/64

The physical devices of the RSU are simpler but the network configuration of each interface is a little more complicated. The DSRC radio unit has the same address for the DSRC interface but also has a *radvd* advertising the prefix for other DSRC nodes that connect to it. The database server is located at addresses `2001:db8:0:3nnn::3/64` and `2001:db8:0:3000::3/64`, where *nnn* is the number of the server.

To make the network easier to use, each network interface in an in-vehicle data acquisition system and RSU is allocated an anycast address. This means that the interfaces share these addresses across all devices. The first to receive a request at an address will answer. Automotive computers are allocated the anycast address `2001:db8::1/64`, DSRC radios `2001:db8::2/64` and database servers `2001:db8::3/64`. The outcome is that a device connecting to the network does not need to know the unit number. For example, by connecting to `2001:db8::1/64`, the device will get the nearest automotive computer. Database servers share the anycast addresses `2001:db8:0:3000::3/64` and `2001:db8::3/64`.

VII. Data Visualization

The software architecture discussed in Section III and the hardware discussed in Sections IV and V allow large volumes

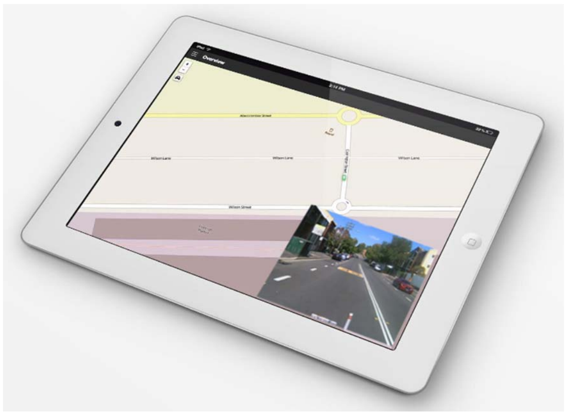


Fig. 7. Live data displayed on a tablet computer including video feed from the front camera.

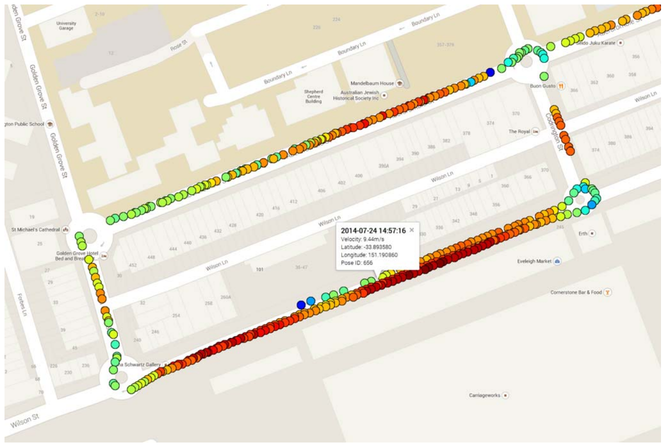


Fig. 8. Geo-spatial analysis showing vehicle speed as a heat map.

of data to be automatically harvested from vehicles. Collecting data from multiple vehicles over an extended period of time has the potential to amass vast quantities of data. Visualizing this big data is an important preliminary step in extracting useful information.

Our architecture is flexible and offers many ways to visualize the data. The system described in this paper permits real-time display of data. This is discussed in Section VII-A. Off-vehicle methods for visualizing data are discussed in Section VII-B.

A. In-Vehicle Visualization

Any mobile device with the capacity to connect to a wireless network is able to access data broadcasts within the *Blackbox*. Mobile devices can connect with the *Blackbox* via the WiFi network it provides. Although this capability is useful for exchanging information between devices, connected mobile devices would require additional software to read and display the data broadcasts. Instead of requiring mobile devices to rely on specialized software, summary data is made available via a website hosted on each *Blackbox*. Once on the *Blackbox* network, mobile devices only require a browser to view live system updates.

The *Blackboxes* also host a PostgreSQL database [19] with GIS extensions [20] installed on it. This allows OpenStreetMap (OSM) data to be stored on-board the vehicle for a local region. This data provides a rich source of contextual information to the wider software ecosystem such as the name of the current street, the speed limit or distance to the next intersection.

An example illustrating a tablet device connected to the vehicle WiFi network is shown in Fig. 7. The device has navigated to the locally hosted website which is serving the current position of the vehicle, the surrounding OSM tiles and an intruder vehicle communicating via DSRC. This sort of visualization allows researchers to coordinate maneuvers in multi-vehicle field trials and confirm that data is being logged properly. Real-time visualization also forms the basis for providing information to the driver to improve situational awareness. Visualization can include information about any threats that the system identifies, such as increased risk of collision or abnormal behavior of other vehicles.

B. Off-Vehicle Visualization

Several methods of off-vehicle visualization are available. The data can be replayed as if it were a live system, as discussed in Section VII-B1. Alternatively, large volumes of data can be mined from the database to provide a rich source for 2D visualization, as described in Section VII-B2. Section VII-B3 describes how historic data can be used to simulate traffic environments.

1) Playback: Real-time playback of historic data provides a repeatable mechanism for testing new algorithms in an artificial environment based on real data. Testing algorithms in this environment allows them to be developed and validated using historic data prior to costly field trials and deployments.

Using inter-process communication (Section III-B) to transfer data between processes has the useful property that for processes receiving messages, there is no difference between live and replayed data. Data consumers have no way of identifying if the data is being generated live, or if historic data is being re-injected into the system. To playback data, a broadcaster can be created for each message type in the system. The logged messages are then read from the log files (Section IV-C) or database (Section V-B) and published via the appropriate broadcaster. The order and timing of broadcasts during playback is structured to replicate the traffic that was originally observed during logging.

2) Database Queries: Over time, the database is able to automatically acquire a large amount of data from multiple vehicles. Aggregating data in one location allows queries to be formed about the performance of individual drivers, types of drivers, geospatial locations or entire transport networks. Traffic flow rates and congestion, hotspots for traffic rule violations, near misses, and network efficiencies can all be reported upon with the richness of data collected.

Geospatially referenced data can easily be reported using a top-view heat map such as in Fig. 8. Even more complex queries such as selecting all data where vehicles stopped within 50 meters of a traffic signal are trivial to implement. The results of this query are shown in Fig. 9.

3) Simulation: Traditional visualization of traffic data is typically performed in either geospatial or temporal domains. Colored maps and time series graphs are common representations

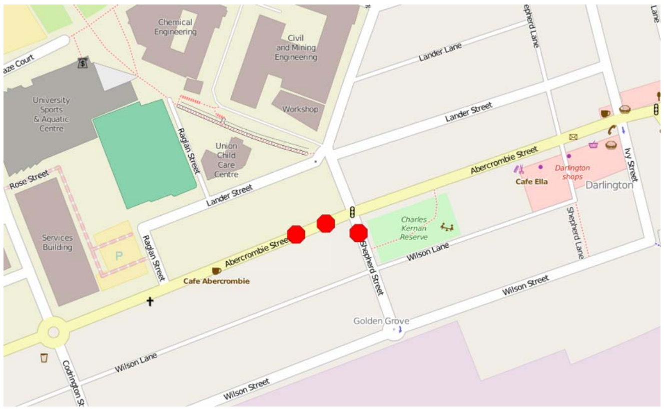


Fig. 9. Geospatial query performed on database to return all positions where a vehicle is stationary within 50 meters of a traffic signal. These positions are shown as red octagonal markers.

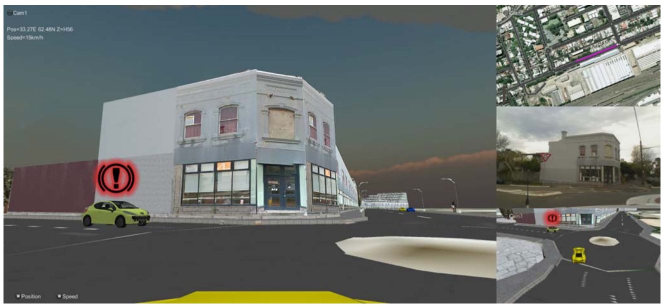


Fig. 10. 3D visualization tool used to replay logged data. Note the annotations that can be added to the scene, and the close correspondence between the simulated environment and the video still (middle, right image) captured during the trial.

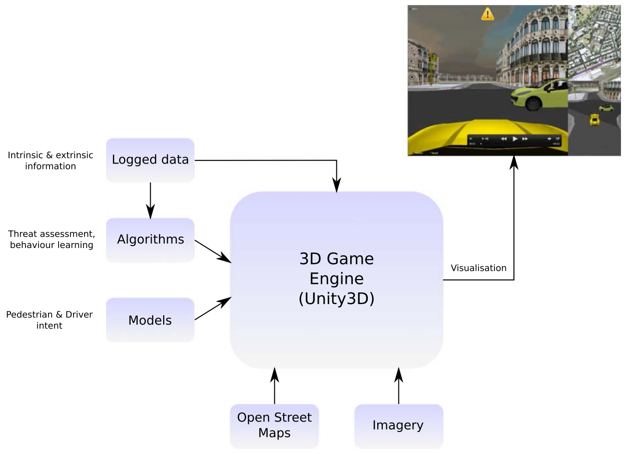

Intrinsic & extrinsic information
Logged data
Threat assessment, behaviour learning
Algorithms
Pedestrian & Driver intent
Models
3D Game Engine (Unity3D)
Visualisation
Open Street Maps
Imagery

Fig. 11. Operational scheme of the proposed visualization tool. The core is a 3D game engine with inputs from the logged data, algorithms, models and maps. The output is a 3D environment with the potential to introduce synthetic agents.

in these two domains [21]. An alternative method for analyzing this information is to unify the geospatial and temporal domains in an immersive, three-dimensional environment. Visualizing the data in this way provides a natural and intuitive method for people to relate to traffic interactions.

Fig. 10 shows work we have done to replay logged data within a 3D engine. Fig. 11 shows the variety of data sources that can act as inputs to the simulation. Logged data is combined with algorithms and agent models to produce a simulated environment. To add realism, geospatial information and street level imagery are added to the scene. Several 3D game engines are capable of implementing this visualization strategy and have been discussed in the literature [22]. In our work, Unity3D is used.

Accurate simulated environments are a useful research tool. On a very basic level, the simulated environment provides a mechanism for visualizing recorded data from any point of view. For example, the data can be viewed from the perspective of any driver or pedestrian in the scene. However, the utility of a simulator is not limited to replaying recorded data. Future simulators will be capable of running synthetic scenarios with simulated agents. To convincingly depict agents in synthetic scenarios, statistically accurate models of driver behavior need to be developed. As large databases of naturalistic driving data are collected, these models will be developed. The ability to create synthetic traffic scenarios will enable researchers to study driver behavior in a safe and controlled environment without the cost and resources required for live testing.

VIII. Evaluation

The foundation of our proposed architecture is the software paradigm described in Section III and the IPv6 communication strategy described in Section VI. A high-level comparison of our proposed software architecture and several other communications libraries is included in Section VIII-A. The performance of our system, relative to the other communication libraries, is quantified in Section VIII-B. A broader evaluation of our software architecture and its application to our research is discussed in Section VIII-C.

A. *Comparison*

Motivating factors in the design of our system were extensibility, ease of maintenance and reliability. We call the communication backbone of our system the multiprocess communications library (MCL) and have made it publicly available [23]. Prior to developing this software we considered LCM [8], ZMQ [24], robot operating system (ROS) and RabbitMQ (RabbitMQ) [9]. A comparison of key differences between the communication libraries is shown in Table II. These libraries can be broadly divided into two categories: centralized and decentralized.

ROS and RabbitMQ are similar in that a central service is needed to facilitate message routing. The problem with these architectures is that the central service forms a single point of failure that can disable communication within the system. While adding new processes or messages is relatively trivial, adding new nodes to the system requires extra configuration. Each machine that joins the network needs to know both a message specification and the location of the central service. In our software architecture, nodes joining the network only need to acknowledge the message specification. They do not need to perform any configuration. By acknowledging each message type is uniquely associated with a multicast address, new nodes can participate by simply joining the network.

ZMQ is a high performance communication library that provides a socket-like interface for transmitting and receiving data. One of our goals was to implement a communication system where many-to-many connections are possible. That is, many broadcasters can publish data on the network to many

TABLE II
COMPARISON OF COMMUNICATION LIBRARIES

	MCL	LCM	RabbitMQ	ROS	ZMQ
Structure	Decentralised	Decentralised	Message Broker	Centralised Directory	Socket like message queue
Transport	UDP	UDP	TCP	TCP	TCP
Topologies	publish/subscribe	publish/subscribe	Many	publish/subscribe	Many

consumers. While ZMQ allows many connections to listen for data on a particular socket, only one process can bind to a socket and publish. To allow a many-to-many topology, complex networks, a directory service or broker need to be implemented. Robustly implementing these solutions is a challenging task.

Our system is most similar to LCM. In LCM, strongly typed messages are broadcast using IPv4 UDP multicasts. In our architecture weakly typed messages are broadcast using IPv6 UDP multicasts. By utilizing multicasts, both libraries are robust in the sense that there is no single point of failure. Rather than relying on brokers or discovery services, multicasts allow any process to listen for or broadcast data on a particular address. If a particular process blocks, it will not impede the ability of the other processes to communicate. A point of difference between the libraries is that our library is capable of transmitting weakly typed messages where any serializable object may be transmitted. While our software allows messages with mandatory fields to be defined, it is also possible to dynamically create and transmit new fields. In LCM, messages are strongly typed making it impossible to dynamically add new fields to a message.

The communication libraries can also be categorized by the protocol used to transmit data. ZMQ, ROS and RabbitMQ use TCP to transmit data. MCL and LCM use UDP to multicast data. TCP is a more robust protocol as it guarantees transmission and order of delivery whereas UDP does not. On the other hand, UDP offers lower latency and a connectionless transmission model. The connectionless transmission model is what allows the UDP based communication libraries to operate in a decentralized manner.

B. Quantitative

Our tests are structured similarly to the tests performed in [8]. A single process transmits *ping* messages at various rates. A number of client processes receive the *ping* messages and echo the data as a *pong* message. A final process is used to log all of the network traffic. The logged data is used to calculate network performance.

Network performance is quantified by both bandwidth and latency. Transmission bandwidth is calculated by comparing the observed number of logged *pings* against the target send rate. This measures the software's capacity to send data at high frequencies. The receive bandwidth is calculated by comparing the observed number of logged *pongs* against the target *ping* send rate. This measures the software's capacity to receive data at high frequencies. Note that the receive rate will be limited by the transmission rate. The time is recorded when each *ping* message is created and transmitted. The time is also recorded when a client receives a *ping* message and copies the data into a *pong* message. These two time-stamps are used to calculate the latency of sending data over the network.

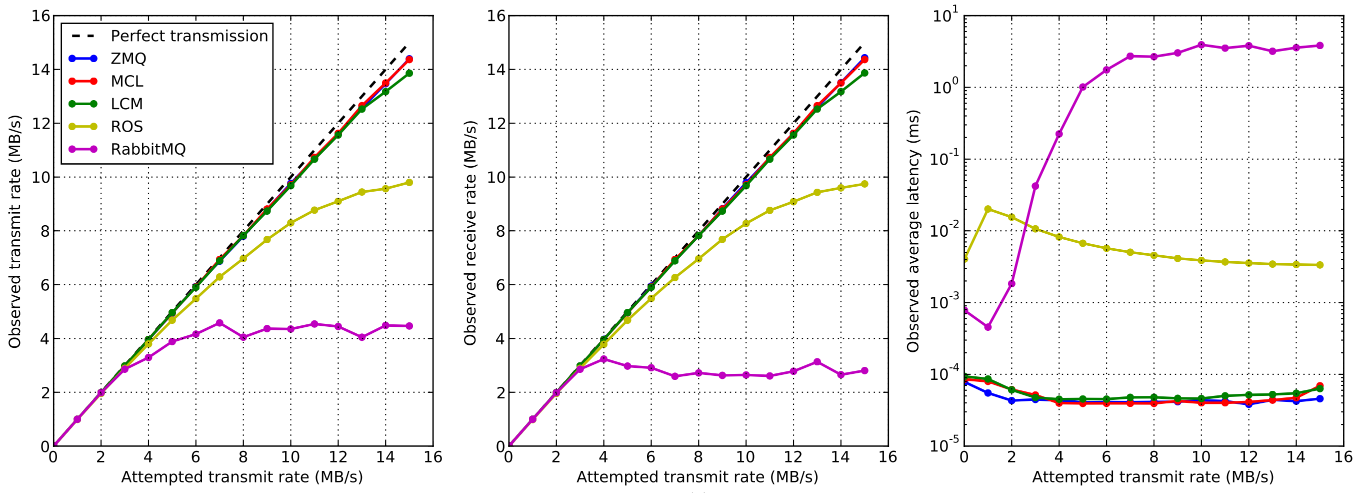
In the following tests ZMQ, LCM, ROS and RabbitMQ are compared to our system. To ensure the implementations are as comparable as possible, an effort was made to standardize the testing code for each communication library. As mentioned earlier, the primary motivating principles in the design of our system are rapid prototyping and ease of development. Following these considerations our software language of choice is Python. We have consciously traded-off the performance of compiled languages like C++ for ease of development. The tests discussed in this section were conducted in Python. The results presented are objective when comparing the communication libraries for use in Python. Care must be taken not to view the results as an absolute statement of performance that applies across all language implementations of each library.

1) Localhost: To test the software's ability to scale with multiple processes, a single machine hosted a *ping* server, a logging process and 1, 3, and 6 *pong* clients. Each *ping* space and *pong* message contains a payload of one kilobyte. The results are averaged over a testing window of 10 seconds. The results are shown in Fig. 12. The data show that our software (MCL) is able to provide comparable bandwidth and latency to both LCM and ZMQ for up to three *pong* clients while still remaining a high performance option up to six *pong* clients. In all scenarios *MCL* is preferable to both *RabbitMQ* and ROS.

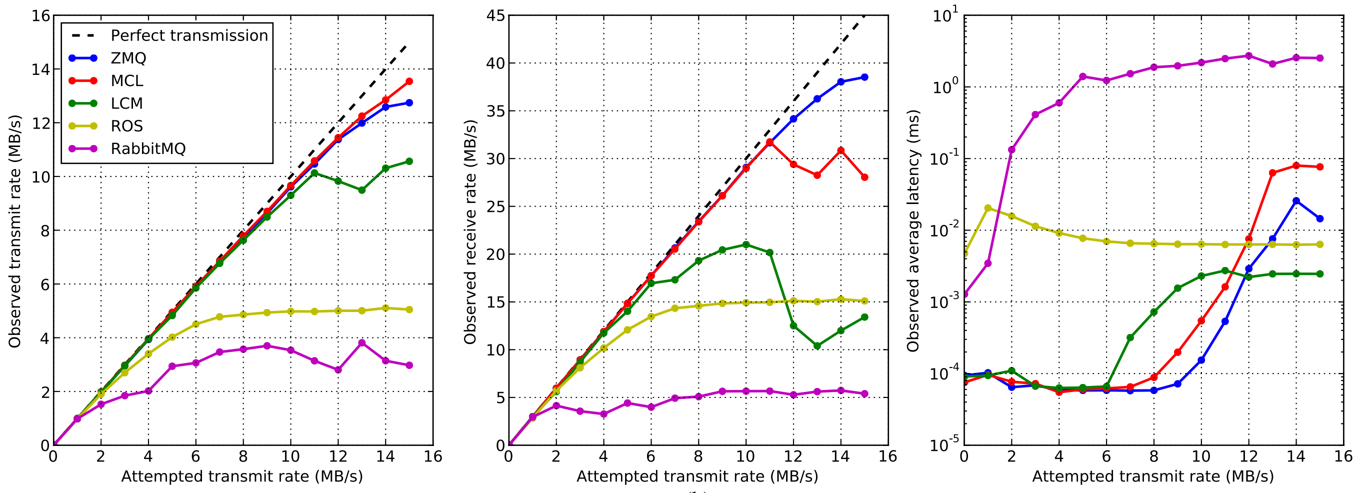
To test the software's ability to scale with message size, a single machine hosts a *ping* server, a logging process and three *pong* clients. Each *ping* and *pong* message contains a payload of 500, 1500 and 3000 bytes. The results are averaged over a testing window of 10 seconds. The results are shown in Fig. 13. The data show that as the payload size is increased, the observed transmit and receive bandwidth increase for all communication libraries. Again, MCL is able to provide comparable bandwidth and latency to both ZMQ and LCM. As the message size increases, all methods converged on equivalent bandwidths with the exception of RabbitMQ.

Figs. 12 and 13 show that as message traffic is increased, both transmit and receive bandwidth deviate further away from perfect transmission. The number of messages sent per second can be increased by introducing more *pong* clients into the system or by decreasing the size of the message payload. Note that as the message payload is decreased, more messages need to be transmitted to realize a target bandwidth.

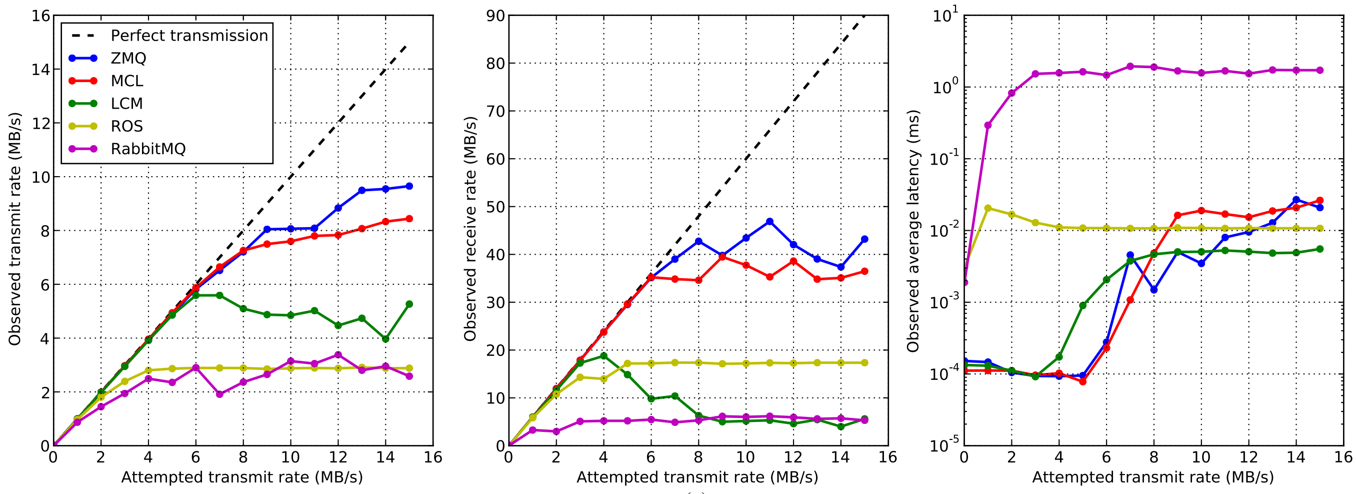
In these tests, the main bottleneck in the system is the amount of available computing power. The tests were performed on a consumer laptop with an Intel i7-2640M processor and 8GB of RAM. As the number of messages sent per second is increased, computing power becomes a scarcer resource. Messages are being received faster than they can be processed. Differences in the implementation of the communication libraries start to affect how much traffic they are able to handle.



(a)



(b)



(c)

Fig. 12. Performance on a single machine using one, three and six *pong* clients. Each message contains a payload of one kilobyte. The results are averaged over a testing window of 10 seconds. The first column shows the observed transmit rate plotted against the target transmit rate. The second column shows the total observed receive rate for all clients plotted against the target transmit rate. The final column shows the observed latency plotted against the target transmit rate.

The converse is also true. As the number of messages in the system is decreased, the communication libraries are largely decoupled from inefficiencies in their implementation and hardware limitations. Evidence of this can be seen in Fig. 13(c) where all communication libraries, with the exception of RabbitMQ, are able to provide similar performance. This is achieved by transmitting large messages at a low frequency.

The results show that the decentralized methods out-perform the two centralized methods. Both ROS and RabbitMQ rely on a centralized process to permit communication. These central

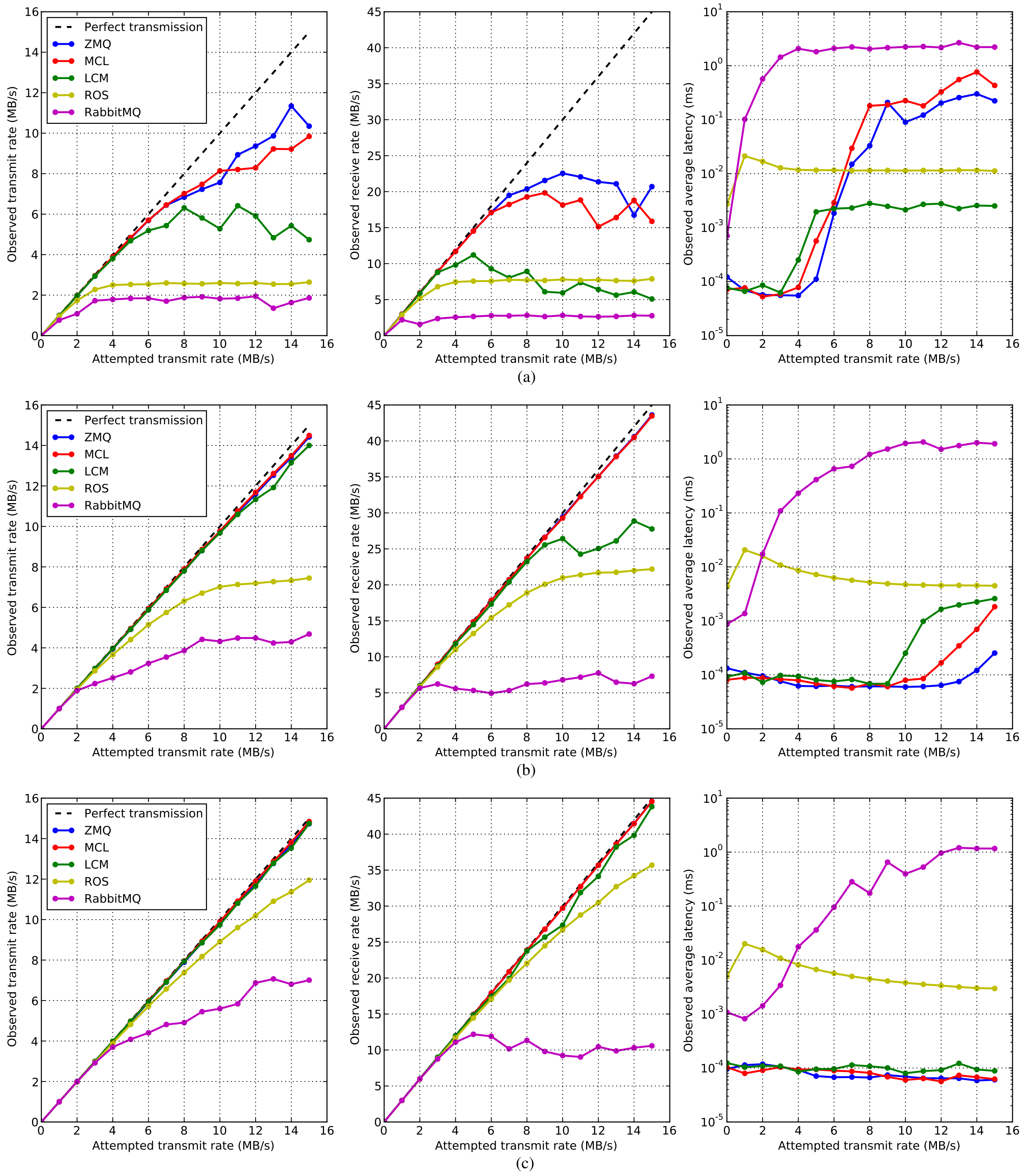


Fig. 13. Performance on a single machine using 500, 1500, and 3000 byte message payloads. Three *pong* clients are included in the test. The results are averaged over a testing window of 10 seconds. The first column shows the observed transmit rate plotted against the target transmit rate. The second column shows the total observed receive rate for all clients plotted against the target transmit rate. The final column shows the observed latency plotted against the target transmit rate.

processes become bottlenecks in the system as the data rate rises. This can be observed in the ROS graphs which plateau when the capacity of the system is reached. The decentralized architectures avoid these bottlenecks by permitting direct communication via sockets (ZMQ) or via multicasts (MCL and LCM). This makes the decentralized strategies more scalable as they are amenable to distributing the processing load over multiple machines.

Whilst *ZMQ* offers the highest bandwidth and lowest latency, it is the least flexible architecture. Since only one process

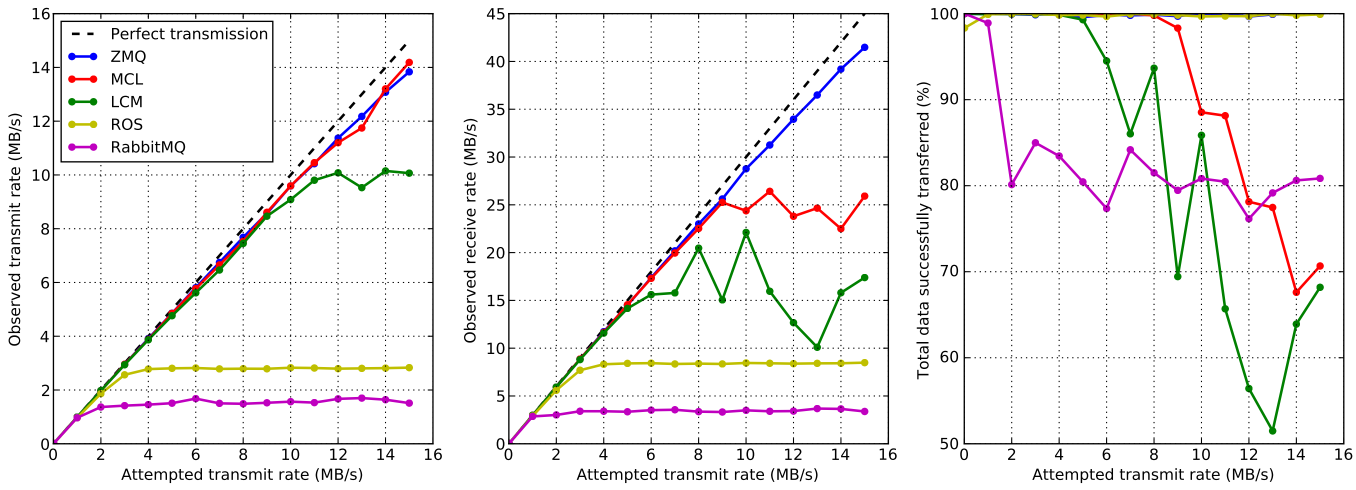


Fig. 14. Performance on a network of machines using a single *ping* server and three *pong* clients. Messages with a 500 byte payload are transmitted. The results are averaged over a testing window of 10 seconds. The first column shows the observed transmit rate plotted against the target transmit rate. The second column shows the total observed receive rate for all clients plotted against the target transmit rate. The final column shows the percentage of total traffic that was observed.

can bind to a socket and publish, each client publishing a *pong* message must open up a new and unique socket. To receive all *pong* messages, a new process will need to know how to connect to each *pong* socket. The tests formed a trivial network where the number of clients was known a priori. This made it possible to "hardcode" the network topology. In complex and dynamic networks this naive strategy does not scale well.

The tests illustrate that MCL is able to provide a good mix of high bandwidth, low latency and scalability with minimal configuration. On a qualitative level, the testing code written using the MCL is more compact than the other libraries. The event driven architecture we advocate in Section III-A allows transparent code to be written in a few lines. We believe this promotes code that is easier to debug, maintain and extend.

2) Network: To test the software's efficiency on a local area network, a test was performed using one *ping* server and three *pong* clients. The laptop used in the previous test was used as a server to issue *ping* messages and log network traffic. Three *Blackbox* (Section IV-B) clients were used to return *pong* messages. All computers were connected via a gigabit switch. The clocks in the *pong* clients were synchronized to a NTP server hosted on the *ping* server. For ROS and RabbitMQ the central processes were hosted on the *ping* server. Each *ping* and *pong* message contains a payload of 500 bytes. The results are averaged over a testing window of 10 seconds. The results are shown in Fig. 14 and can be compared to the same test performed on a single machine [Fig. 13(a)].

The most notable outcome of this comparison is that the centralized communication libraries (ROS and RabbitMQ) perform similarly whether they operate on a single machine or as part of a distributed network. Conversely, the decentralized methods (ZMQ, MCL, and LCM) benefit significantly from operating in a distributed network. In a distributed network, the decentralized methods are able to take advantage of additional nodes by sharing the processing load across the machines.

The latency of the communication libraries could not be measured reliably. Although the clocks were synchronized via a NTP server, each library was able to send data with a latency smaller than the jitter in the NTP synchronized clocks. Instead of showing latency (as in Figs. 12 and 13), the last sub-plot in Fig. 14 shows the total percentage of data that was successfully transferred. This is measured by calculating the ratio between the number of *pings* and *pongs* that were observed against the number of *pings* and *pongs* that were sent.

The total percentage of data that was successfully transferred shows that TCP based methods (ZMQ, ROS) outperform UDP based methods (MCL, LCM). Once the UDP based methods become limited by the processor, they are able to continue operating by dropping packets. While this enables a high transmit and receive bandwidth, it also mean much of the data is lost. Since TCP guarantees delivery, messages cannot be dropped silently. This limits the amount of network traffic that can be exchanged to the capacity of the processor. As a result, the transmit and receive bandwidths are low but data is not lost.

C. Research Utility

A similar architecture for on-board logging and harvesting was used by our group in the mining industry, resulting in a publicly available dataset spanning three years involving over 30 vehicles [25]. To give some idea of the scale of this dataset, a typical day of data contains vehicles driving a total distance of around 3000 kilometers. This dataset has been used in a number of different types of analysis including road mapping [26], long term prediction estimation [27], fault detection [28] and behavior [29] analysis. To the authors' knowledge this is one of the most comprehensive Cooperative ITS datasets available to researchers.

The architecture proposed in this paper improves on our previous data acquisition framework. It is more automatic, scalable, flexible and reliable. This makes it an ideal environment for rapid prototyping and experimenting with new sensors and algorithms. The volume of data our proposed architecture is able to collect automatically also makes it an ideal environment for gathering big data.

The system presented in this paper has been deployed at our test site. Two research vehicles are fitted with data acquisition systems (Section IV). Two RSU (Section V) are located on the campus. One RSU is located in a position we drive near frequently. This allows us to test live-streaming capabilities. The second RSU is located near the vehicle parking bays. This ensures all data can be uploaded automatically after a drive has concluded. The process of collecting data is simply driving the vehicle and parking it in the designated parking bays at the end of the drive. The data can then be accessed by querying our database. This low intervention approach has greatly simplified our ability to perform experiments.

The system has been running and collecting data without human assistance for a year at the time of submission of this document. Although our current deployment has been operating for a shorter period of time and on fewer vehicles than [25], it has currently automatically logged several million data. In that short time, the proposed architecture has fulfilled its design criteria. Data collection and mining is trivial due to the automatic collection infrastructure. The flexibility of the software has also permitted new algorithms to be integrated into the system. As a result we have been able to focus our efforts on performing research on driver behavior [30], [31] and navigation filter consistency [32].

IX. Conclusion

This paper introduced a modular architecture for the acquisition, storage, processing and visualization of data in ITS applications. The design is based on a publish–subscribe paradigm using IPv6 multicast for inter-process communication. This strategy allows code to be broken up into modular designs that can be run on many independent processes. Message passing over the publish–subscribe network allows communication to occur transparently within a single computer, across multiple computers and across heterogeneous platforms running different programming languages.

The software architecture described is used for both in-vehicle data acquisition and off-vehicle data acquisition. The in-vehicle data acquisition system automatically records vehicle data streams to log files. When in range of a RSU the in-vehicle data acquisition system is able to automatically upload the log files to a central server via DSRC radio. Once the log files have been uploaded, they are inserted into a database. The combined software and hardware design allows complex data to be recorded from vehicles and inserted into a database with no human intervention. The automatic nature of this design makes the system ideal for gathering big data.

When compared to ZMQ, LCM, ROS and RabbitMQ we show that our proposed architecture offers a good mix of high-bandwidth and low-latency transmission. Due to the design of the system it is decentralized, robust to failure and amenable to scaling across multiple machines with minimal configuration. We believe that the design paradigms advocated in our architecture encourage code development that is transparent, extensible and easy to maintain.

References

[1] J. Ziegler *et al.*, "Making bertha drive—An autonomous journey on a historic route," *IEEE Intell. Transp. Syst. Mag.*, vol. 6, no. 2, pp. 8–20, Summer 2014.

[2] S. Kammel *et al.*, "Team AnnieWAY's autonomous system for the 2007 DARPA Urban Challenge," *J. Field Robot.*, vol. 25, no. 9, pp. 615–639, 2008.

[3] S. Thrun *et al.*, "Stanley: The robot that won the DARPA grand challenge," *J. Field Robot.*, vol. 23, no. 9, pp. 661–692, Sep. 2006.

[4] H. Stubing *et al.*, "simTD: A car-to-X system architecture for field operational tests," *IEEE Commun. Mag.*, vol. 48, no. 5, pp. 148–154, May 2010.

[5] P. Alexander, D. Haley, and A. Grant, "Cooperative intelligent transport systems: 5.9-GHz field trials," *Proc. IEEE*, vol. 99, no. 7, pp. 1213–1235, Jul. 2011.

[6] H. Cai, X. Jia, A. S. Chiu, X. Hu, and M. Xu, "Siting public electric vehicle charging stations in Beijing using big-data informed travel patterns of the taxi fleet," *Transp. Res. D, Transp. Environ.*, vol. 33, pp. 39–46, Dec. 2014.

[7] L. Gong, X. Liu, L. Wu, and Y. Liu, "Inferring trip purposes and uncovering travel patterns from taxi trajectory data," *Cartogr. Geograph. Inf. Sci.*, vol. 43, no. 2, pp. 1–12, 2016.

[8] A. Huang, E. Olson, and D. Moore, "LCM: Lightweight communications and marshalling," in *Proc. IEEE/RSJ Int. Conf. IROS*, Oct. 2010, pp. 4057–4062.

[9] RabbitMQ. [Online]. Available: http://www.rabbitmq.com/

[10] M. Quigley *et al.*, "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, 2009, pp. 1–6.

[11] D. Geronimo, A. Lopez, A. Sappa, and T. Graf, "Survey of pedestrian detection for advanced driver assistance systems," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 7, pp. 1239–1258, Jul. 2010.

[12] G. Agamennoni, J. Nieto, and E. Nebot, "A Bayesian approach for driving behavior inference," in *Proc. IEEE IV*, Jun. 2011, pp. 595–600.

[13] S. Sivaraman and M. Trivedi, "Towards cooperative, predictive driver assistance," in *Proc. IEEE 16th Int. ITSC*, Oct. 2013, pp. 1719–1724.

[14] R. Adla, N. Al-Holou, M. Murad, and Y. Bazzi, "Automotive collision avoidance methodologies sensor-based and ITS-based," in *Proc. IEEE AICCSA*, May 2013, pp. 1–8.

[15] J. R. Ward, G. Agamennoni, S. Worrall, A. Bender, and E. Nebot, "Extending time to collision for probabilistic reasoning in general traffic scenarios," *Transp. Res. C, Emerging Technol.*, vol. 51, pp. 66–82, Feb. 2015.

[16] S. Worrall and E. Nebot, "Automated process for generating digitised maps through GPS data compression," in *Proc. Australasian Conf. Robot. Autom.*, 2007, pp. 1–6.

[17] J. Ward, S. Worrall, G. Agamennoni, and E. Nebot, "Comprehensive data collection and context based metric evaluation for safety monitoring," in *Proc. 16th IEEE ITSC*, Oct. 2013, pp. 658–663.

[18] Y. Ding, H. Tan, W. Luo, and L. M. Ni, "Exploring the use of diverse replicas for big location tracking data," in *Proc. IEEE 34th ICDCS*, 2014, pp. 83–92.

[19] PostgreSQL. [Online]. Available: http://postgresql.org/

[20] PostGIS. [Online]. Available: http://postgis.net/

[21] N. Adrienko and G. Adrienko, "Spatial generalization and aggregation of massive movement data," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 2, pp. 205–219, Feb. 2011.

[22] A. Gregoriades, C. Florides, V. Lesta, and M. Pampaka, "Driver behaviour analysis through simulation," in *Proc. IEEE Int. Conf. SMC*, Oct. 2013, pp. 3681–3686.

[23] Multiprocess communications library. [Online]. Available: https://github.com/acfr/mcl

[24] ZeroMQ. [Online]. Available: http://zeromq.org/

[25] J. Ward, S. Worrall, G. Agamennoni, and E. Nebot, "The Warrigal dataset: Multi-vehicle trajectories and V2V communications," *IEEE Intell. Transp. Syst. Mag.*, vol. 6, no. 3, pp. 109–117, Fall 2014.

[26] G. Agamennoni, J. Nieto, and E. Nebot, "Robust inference of principal road paths for intelligent transportation systems," *IEEE Trans. Intell. Transp. Syst.*, vol. 12, no. 1, pp. 298–308, Mar. 2011.

[27] M. Shan, S. Worrall, F. Masson, and E. Nebot, "Using delayed observations for long-term vehicle tracking in large environments," *IEEE Trans. Intell. Transp. Syst.*, vol. 15, no. 3, pp. 967–981, Jun. 2014.

[28] S. Worrall, G. Agamennoni, J. Ward, and E. Nebot, "Fault detection for vehicular ad-hoc wireless networks," in *Proc. IEEE IV*, Jun. 2013, pp. 298–303.

[29] G. Agamennoni, S. Worrall, J. Ward, and E. Nebot, "Automated extraction of driver behaviour primitives using Bayesian agglomerative sequence segmentation," in *Proc. IEEE 17th Int. ITSC*, Oct. 2014, pp. 1449–1455.

[30] A. Bender, G. Agamennoni, J. Ward, S. Worrall, and E. Nebot, "An unsupervised approach for inferring driver behavior from naturalistic driving data," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 6, pp. 3325–3336, Dec. 2015.

[31] A. Bender, J. Ward, S. Worrall, and E. Nebot, "Predicting driver intent from models of naturalistic driving," in *Proc. IEEE 18th ITSC*, Sep. 2015, pp. 1609–1615.

[32] S. Worrall, J. Ward, A. Bender, and E. Nebot, "GPS/GNSS consistency in a multi-path environment and during signal outages," in *Proc. IEEE 18th Int. ITSC*, Sep. 2015, pp. 2505–2511.

Asher Bender received the Bachelor of Engineering (Mechatronics) and the Doctor of Philosophy degrees from The University of Sydney, Sydney, Australia, in 2008 and 2013, respectively. He is a Research Associate with and a member of the Intelligent Vehicles and Safety Systems Group, Australian Centre for Field Robotics, The University of Sydney. His research focuses on applying machine learning to solve problems in field robotics.

James R. Ward received the Bachelor of Engineering (Aeronautical) and the Master of Education degrees from The University of Sydney, Sydney, Australia, in 1999 and 2003, respectively, and the Ph.D. degree in mechatronic engineering from The University of New South Wales, Kensington, Australia, in 2009. He is a Researcher with the Intelligent Vehicles and Safety Systems Group, Australian Centre for Field Robotics, The University of Sydney. His research interests are in safety analysis, vehicle safety systems, and intelligent transportation.

Stewart Worrall received the Ph.D. degree from The University of Sydney, Sydney, Australia, in 2009. He is currently a Research Fellow with the Australian Centre for Field Robotics, The University of Sydney. His research is focused on the study and application of technology for vehicle automation and improving safety.

Marcelo L. Moreyra (M'02) received the Electronic Engineering degree from Universidad Nacional del Comahue (UNCo), Neuquén, Argentina, in 2007 and the Ph.D. degree in control systems from Universidad Nacional del Sur (UNS), Bahía Blanca, Argentina, in 2013. He is an Assistant Professor with UNCo. His main research interests are in visual perception systems, driver assistance systems, and multisensor fusion for localization and guidance of ground and aerial intelligent vehicles. He is a member of the IEEE Robotics and Automation (RAS) and the Intelligent Transportation Systems societies. He was an SAC Chair of Argentina IEEE Section (2009–2011) and a RAS-SAC Cochair in 2011. Since 2015, he has been the Treasurer of IE13/CS23/RA24/IA34/PEL35/VT06 IEEE Argentina Joint Chapter.

Santiago Gerling Konrad received the Electronic Engineering degree from Universidad Nacional del Sur (UNS), Bahía Blanca, Argentina, where he is currently a Ph.D. candidate. He holds a scholarship from Consejo Nacional de Investigaciones Cientificas y Técnicas (CONICET) at Instituto de Investigaciones en Ingeniería Eléctrica (IIIE) "Alfredo Desages", Departamento de Ingeniería Eléctrica y Computadoras, Universidad Nacional del Sur.

Favio Masson received the B.Sc. degree in electrical engineering and the Ph.D. degree from Universidad Nacional del Sur (UNS), Bahía Blanca, Argentina. He was a Researcher with National Scientific and Technical Research Council (CONICET), Buenos Aires, Argentina. He is a Professor with UNS. His main research interests are in field robotics automation and sensors network.

Eduardo M. Nebot received the B.Sc. degree in electrical engineering from Universidad Nacional del Sur, Bahía Blanca, Argentina, and the M.Sc. and Ph.D. degrees from Colorado State University, Fort Collins, CO, USA. He is currently a Professor with The University of Sydney, Sydney, Australia, and the Director of the Australian Centre for Field Robotics, The University of Sydney. His main research interests are in field robotics automation. The major impact of his fundamental research is in autonomous systems, navigation, and safety.