Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2013, Article ID 415297, 29 pages
http://dx.doi.org/10.1155/2013/415297

*Research Article*

# PLUGAM: Plug-In-Based Adaptation Middleware for Network Layer Protocols for Vehicular Ad Hoc Networks

**Carlos F. Caloca,[1,2] J. Antonio García-Macías,[1] and Thierry Delot[2]**

[1] *Department of Computer Science, CICESE Research Center, Carretera Ensenada-Tijuana, No. 3918 Zona Playitas, 22860 Ensenada, BC, Mexico*
[2] *LAMIH UMR CNRS 8201, University of Valenciennes et Haute Cambresis, Le Mont Houy, 59313 Valenciennes, France*

Correspondence should be addressed to J. Antonio García-Macías; jagm@cicese.mx

Received 4 January 2013; Revised 27 April 2013; Accepted 24 May 2013

Academic Editor: Danny Hughes



Vehicular communication environments are characterized by highly mobile nodes, frequent topology changes, and a great variation in the number of neighbor vehicles. The network layer (NL) protocols must adapt continuously to these unreliable conditions, hence the growing effort in the development of protocols specific to vehicular networks. The aim of this work is to help create vehicular systems with an adaptive network layer, by means of proposing a middleware based on an adaptation model and using as input the context information of the vehicle. The architecture will build an adaptive network layer using two types of run-time adaptations: modification of NL protocol internal parameters and adaptations where a selection between NL protocols must be made. To assess the feasibility of our approach, we present two case study examples and present a first prototype. In addition, we briefly evaluate the adaptation presented in case study 1 via simulation, and we found that it produces 20% less control overhead. Furthermore, we present an analysis of the prototype performance to get a rough idea on the cost of using the middleware in the vehicular system; results show that it is feasible to implement the middleware in hardware similar to today's midrange smartphones.

## 1. Introduction

Intelligent transportation system (ITS) [1] aims to bring significant improvement in transportation system performance, including reduced congestion, increased safety, and traveler convenience. Vehicular ad hoc networks (VANETs) are the most important component of an ITS, in which vehicles are equipped with some short-range wireless communication. VANETs represent a rapidly emerging, particularly challenging class of mobile ad hoc networks (MANETs). In the recent years, there has been intensive research work around VANETs due to the wide variety of services or potential applications they could provide.

Vehicular communication environments are characterized by highly mobile vehicles, extremely frequent topology changes, and a great variation in the number of vehicles in a certain region [2, 3]. These and other dynamic variations in network conditions make the design of network layer (NL) protocols difficult and complicate the development of applications that require efficiency under the dynamicity of these network conditions. Hence, routing protocols must adapt continuously to these unreliable conditions, thus the growing effort in the development of communication protocols which are specific to vehicular networks. Schoch et al. [2] mention some key VANET network conditions like network density, node speed, node heterogeneity, and movement patterns; the authors highlight the extremely opposite values that these network conditions can get in the course of the network's lifetime; for example, the node velocity may range from zero for stationary road side units, or when vehicles are stuck in a traffic jam, to over 130 kph on highways. For us, these characteristics can be seen as context information useful for the network layer protocol.

Some authors [2, 4] state that potential vehicular applications require different communication or types of network layer protocols that exist in the literature to optimize communication. For example, applications like postcrash or breakdown warnings are adequate to use geobroadcast or information dissemination protocols, because the information has to be delivered as quickly as possible to a subset

of surrounding vehicles. On the contrary, applications like Internet connectivity and road surface conditions to traffic authority center are not well suited to use dissemination protocols, because the destination of the message is of great importance and one-to-one communication is desired; in this case, using unicast routing network layer protocols seems more adequate.

We think that having a vehicular system that supports most of these potential applications will help promote the adoption of a real world VANET technology by offering a wealth of services that encourages the driver to go premium and get the on-board system in their new vehicle or mitigate the investment of buying and installing the on-board module in their current vehicle. Nevertheless, for this to work, each application underlying NL protocol must maintain a good performance in spite of the changing network or even applications conditions. There could even be a case in which for an application to maintain good service performance, adapting the NL protocol is not enough and requires the use of different types of NL protocols.

In reference to the applications requiring different communication protocols, authors like Schoch et al. [2] have proposed having a network stack with a set of different network protocols to satisfy all the different applications needs. Schoch et al. also proposed a set of NL protocol types (referred to by them as communication patterns) that they think will be enough to support most of the vehicular applications in the literature. Some of these protocol types are dissemination, unicast, and carry and forward. Some other authors have made different categorizations of NL protocols, for example, in [4, 5]. From the point of view of integration of the vehicular systems of the automotive makers and the research and academic projects, we think that at least two NL protocols are needed in the vehicular system: a unicast NL protocol to enable V2I communication for the infoentertainment applications and a dissemination NL protocol to enable V2V communication for security on the road applications.

Exploring the state of the art in NL protocols in VANET, we notice that the dynamic aspect problem of a VANET is too vast and proposals only attack part of the problem. These proposals focus on adapting to specific context information or network conditions, optimize for specific scenarios like highway, city, low density network, and so forth, and are tailored for using a specific NL protocol. In addition, even when all the proposals solve the dynamic network problem using some kind of adaptation solution, they all use different concepts and terminology to describe their solution. For example, we can find proposals for different types of protocols using concepts like hybrid, adaptive, context-aware, reflective, autonomic, self-managed, policy-based, and so forth. This use of different concepts and terminologies complicates the comparison between proposals, reuse, and generalization the adaptation ideas. Another observation of the state of the art is that we identify a need for vehicular systems using more that one type of NL protocol to support more vehicular applications instead of trying to find one that serves for all. Moreover, we think that there could be situations in which the application can select what NL protocol to use depending on the current context or to switch to using a different-type NL protocol when the current context makes the default NL protocol perform poorly.

Based on the previous problem context, we have the strong conviction that a future network layer that could truly satisfy the needs of potential vehicular applications and deal with the dynamicity of a VANET must have the following functionality:

(i) support multiple types of network layer protocols, in addition, allow adaptations where there could be a selection on which protocol to use based on context decisions or switch to using another NL protocol when the current context generates poor performance,

(ii) have network layer protocols with the ability to adapt their internal parameters based on the current context.

That is why the focus of the present work is to propose a middleware architecture to help create vehicular systems with an adaptive network layer. This architecture is a generic approach as it will build a subset of adaptive NL protocols found in the literature, and because the architecture must be instantiated by adding the modules to provide the context information, the NL protocols and vehicular applications are required to build the desired vehicular system. The architecture will be able to build an adaptive network layer using in general two types of run time adaptations: modification of internal parameters of an NL protocol and adaptations where a selection between NL protocols must be made; both of these adaptations will use as input the context information of the vehicle. The core ideas of our adaptation middleware architecture can be summarized in the following points.

(i) Propose a model of the adaptation concept.

(ii) Propose a design methodology to build the adaptations in the architecture by dividing the solution into submodules (components) which are independent of each other, and then combine them by describing their relationships to form the adaptive network layer.

(iii) Propose a multiple applications-multiple NL protocols environment.

The rest of the paper is organized as follows: we start by describing the adaptation model used by the architecture. After that in Section 3, we present in detail the architecture for the proposed adaptation middleware. Then, in Section 4, we present some case studies examples and how they are implemented in our architecture. In Section 5, we describe how we implement a first prototype of the adaptation middleware. We proceed in Section 6 with the evaluation of our adaptation middleware which focuses on evaluating the performance of one of our case studies examples and presenting an analysis of the cost and performance of the prototype. Then, we present the related work in adaptive NL protocols and other architectures that promotes adaptation. Finally, Section 7 contains conclusions and possible future work.

## 2. The Adaptation Model

Prior to the description of the adaptation middleware architecture, we will present how we modeled adaptations with a focus for NL protocols; this adaptation model will be the core of the architecture.

The adaptation concept refers to the ability of a process or entity to dynamically alter its behavior during its lifetime depending on the changing conditions of the environment. In computer software, the adaptation term appears in subjects like adaptation of the video quality depending on the network connection [15], adaptation of the GUI of an application based on the display resolution of the device [16], reconfiguration of the components forming a system [17, 18], and so forth. There are also some other topics in computer science that center on the idea of adaptation, for example, the term of computational reflection, dynamic aspect oriented programming, policy middleware, and autonomic computing. In our work, we refer to the adaptation of the network layer of a vehicular system; this means that the process or entity that will be adapted is the NL protocol.

In regard to adaptations in communication protocols, Grace [19] distinguishes between two distinct classes of adaptations: in node-local adaptation, the adaptation triggered by a node only involves changing the behavior of itself. In distributed adaptation, an adaptation triggered by a node involves changing the behavior of itself and other nodes in the network. An example of a distributed adaptation is the dynamic reconfiguration of sensor networks proposal of Grace [19]. Distributed adaptations are more complex because they have additional challenges to take into account; for example, proposing a consensus mechanism for situations when multiple nodes perform a distributed adaptation with different results due to the different local context information. Other additional challenges are the need for a notification and scheduling protocols in order to perform the adaptation operation in all participants. Finally, special care is needed to avoid performing conflicting adaptations that can render unexpected results. In the case of our work, this version of the architecture is only focused on dealing with node-local adaptations; extending the work to support distributed adaptations will be considered as future work.

Grace [19] also identifies two kinds of general approaches to software adaptation: parameter adaptation and compositional adaptation. Parameter adaptation modifies predefined program variables that determine a systems behavior; there is no introduction of new algorithms and behavior. Compositional adaptation allows the dynamic recomposition of software modules to change or introduce new behavior. Our adaptation middleware architecture falls into the parameter adaptation approach of software adaptation.

After describing the adaptation concept, we proceed to the description of the adaptation model. Foremost, from now on we will refer to any NL protocol proposal that adapts in this manner, even previously proposed works, by the name of "adaptive protocol." The adaptations inside these adaptive protocol proposals take as input the local context information from the vehicle.

A key point of the adaptation model is the idea of separating the functionality of an adaptive protocol into two independent parts. One part is the network layer functionality of a protocol, for example, the definition of messages format, message sending/receiving operations, suppression mechanism for repeated messages, topology update, connection to upper and lower layers, and so forth. We think that this is the biggest part of an adaptive protocol because the development of a network layer protocol is very complex. The second part is the functionality that performs the specific adaptations in the protocol that enhance it and make it more dynamic; we named a specific adaptation in the protocol as an "adaptation solution"; an adaptive protocol could have multiple adaptation solutions that enhance the protocol.

This idea relies on the separation of concerns principle [20], a general principle in software engineering that promotes the separation of different interests or concerns in a problem, solving in separate modules without requiring detailed knowledge of the other parts, and finally combining them into one result. For us, the two concerns to separate are the adaptation solution functionality and the network layer protocol functionality, and by solving them separately our model will offer a mechanism to combine them to form a complete solution which in our case is the adaptive protocol.

The adaptation model also proposes the separation of the adaptation solution concept into subconcepts (named by us as adaptation elements) and the description of their relationship. Applying again the principle of "separation of concerns," the adaptation solution is divided into three elements.

(i) *Context element:* it symbolizes a piece of information about the environment which is used as input to produce an adaptation solution. This concept also encapsulates the instructions or codes that manage and provide this context information in order to be used for the adaptation. This is similar to the concept of metric from the works of Boleng et al. [21] and Yawut et al. [22]; however, we decided to use the simpler word element instead of metric.

(ii) *Adaptation algorithm:* it encapsulates the instructions, logic, or operations that use the adaptation solution, which is based on the context elements to produce an adaptation as a result. This is also similar to the concept of "policy" used by Hadzic et al. [23].

(iii) *Adaptation action:* it symbolizes a characteristic, element, or action that the adaptation solution modifies, activates/deactivates, or does as a result of adaptation solution. This is similar to the concept of "mechanism" of Hadzic et al. [23].

The separation of the adaptation solution in these three concepts further promotes the modularization of the adaptive protocol. A second benefit is that it facilitates the comparison and classification of prior works by observing the adaptation elements used by them. Finally, this separation could enable a rapid calibration of an adaptation by simply exchanging the adaptation elements in it until a right combination produces a desired result.

The relationship between the elements of an adaptation solution is modeled as an algorithm with inputs and outputs:

$$\text{Algorithm } (N \text{ Context Element Values}) \longrightarrow R \text{ Action Values}. \quad (1)$$

The adaptation algorithm takes as input a list of context elements, their values are used by the instructions and operations inside the algorithm, and the current values of the context elements are collected from the context environment when the input list is constructed. The adaptation algorithm produces as a result a list of values for the associated adaptation actions. Finally, these values are transmitted to the adaptation actions entities that modify the behavior of the adaptable process based on these values.

Another aspect of the adaptation model is the definition of where or when in the process will an adaptation solution be executed. We named these places or points of the process as "adaptation modes." For example, for an NL protocol, some places or points to execute adaptation solutions could be when a node receives a message, when the routing table is modified, every $T$ seconds, when some context elements conditions are met, and so forth. Our adaptation model defines a set of adaptation modes which are specific for each type of process wanting to adapt. Hypothetically, a more general and flexible adaptation model could allow adaptation modes to be in every point/place in the process (e.g., in [17]); however, we think this entails changing the way the original process works internally to be able to support this functionality. In the case of network layer protocols, a complete reimplementation of a protocol will surely be required, and this is not an easy task. That is why we opted to having only a set of adaptation modes, in hope that a proper selection of the adaptation modes can surely diminish the task of porting an already implemented NL protocol in the literature.

The last aspect of our adaptation model is how the network layer protocols are integrated into the adaptation solution concept and its elements. The NL protocol is tied to the adaptation action concept; an NL protocol will offer a number of adaptation actions to allow adaptation solutions to change its behavior via changing the value of the adaptation action. Moreover, there will be a handful of special adaptation actions that are not associated with a specific NL protocol but instead are offered by our middleware; we will talk more about them in later sections. In Figure 1, we show the relationship of all these adaptation elements described in an entity/relationship model.

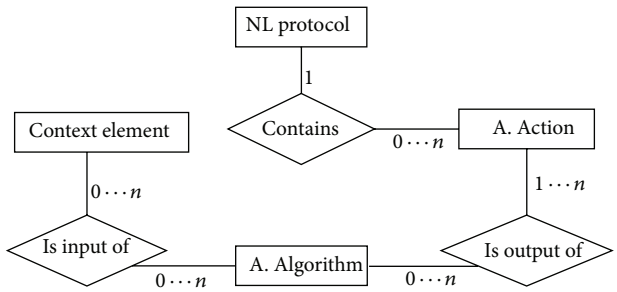


Figure 1: Entity/relationship model of the relationship of the elements of the adaptation solution.

## 3. PLUGAM Architecture

In this section, we will describe the architecture of the proposed adaptation middleware called PLUGAM (Plug-in-based adaptation middleware), this architecture is based on the adaptation model described in the previous section. This description will be guided by explaining the key aspects of the architecture; first we will present an overview of the PLUGAM architecture and the details will be described in the following subsections.

The objective of the PLUGAM middleware is to propose a generic architecture to build a set of adaptive protocols. The adaptation solutions embedded in the adaptive protocols are built using external and independently developed components, called "extension components" in the architecture, loaded in the middleware at startup. The extension components defined in the architecture are the adaptation elements (context elements, adaptation actions, and algorithms), the applications, and NL protocols. After loading all the extension elements needed to build the desired adaptation solution, the next step is to express the combination of the extension components forming an adaptation solution, that is, the adaptation algorithm used, the context elements that will add the inputs to the algorithm, the adaptation actions that will be tied to the outputs of the algorithm, and in what adaptation mode will the adaptation solution be executed, among other things. To express this combination, an "adaptation configuration" must be created in the middleware (see Section 3.6 for more information).

The architecture makes use of plug-in technology as a mechanism to encapsulate and load the extension elements to the middleware; the relationship between the different types of plug-ins and the extension elements offered by these plug-ins is described in Section 3.4. Our proposed architecture also includes a mechanism to uniquely identify an extension component from all others, even from others of the same type (e.g., between NL protocols or context elements). The main use of this identification mechanism is to reference what extension component is required in each field of the adaptation configuration, among other uses that we will see in detail in Section 3.5.

As mentioned in the adaptation model section, an adaptation mode is a place or point of an NL protocol where an adaptation solution is executed (see Figure 2). We proposed to use four adaptation modes: send, receive, forward, and CUPIB. The send, receive, and forward modes are in reference to the events between an application and NL protocol; for example, send mode refers to the event when the application calls the send command and the message must be directed to the protocol. The CUPIB mode is not tied to an application and NL protocol event, but rather is a mode that executes the adaptation solution at a certain period of time. A detailed description of these adaptation modes can be consulted in Section 3.3.

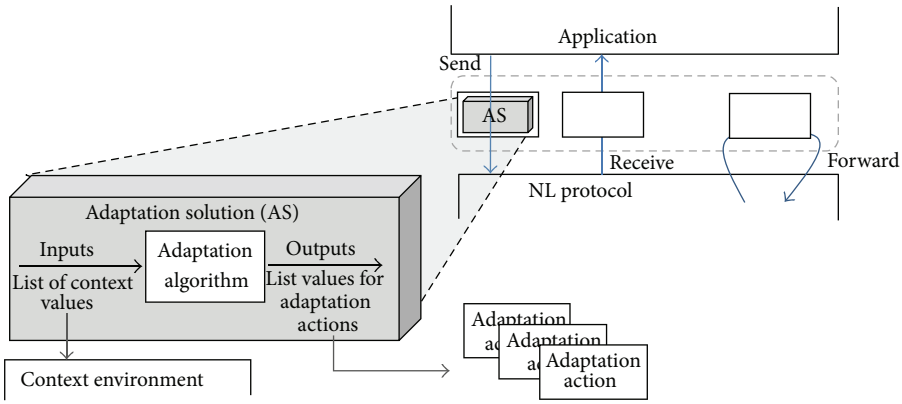


Figure 2: PLUGAM architecture graphical representation.

As shown in Figure 2, we modeled an adaptation solution as a module containing an algorithm, which receives as input a list of context elements values and produces as output a list of values used to execute adaption actions which are parameterized with these values. Before executing an adaptation solution, the architecture will retrieve the current context element values from a context environment defined in the architecture. This context environment is made by different context sources offering context elements; more details on these context sources and the definition of a context element can be found in Section 3.2. Referring to the adaptation actions, apart from the adaptation actions offered by the NL protocols, the architecture offers two special adaptation actions: the "switcher" and the "filter"; more details about these special adaptations actions are presented in Section 3.8.

Finally, in this architecture the network layer is composed of multiple NL protocols, all of them executing concurrently at run time; this is different from a common network stack in which there can be multiple applications in the application layer but in the network layer there is only one protocol. In this multiple NL protocol environment an application needs to subscribe to multiple NL protocols at initiation; then at run time, it can send and receive messages from these NL protocols. A more in-depth description of the role of applications in the architecture can be seen in Section 3.9.

This concludes the overview of the PLUGAM architecture; from here on we will describe the details of the architecture, starting with the characterization of the intended user of the middleware.

*3.1. The User of PLUGAM Middleware.* This middleware is intended to be used by the automotive maker developer which is building the on-board vehicular computer with VANET capabilities, 3G, and so forth. It is not intended to be used directly by the driver. This middleware will be a development tool to help the automotive maker develop more dynamic vehicular applications by having an adaptive network layer in the vehicular system. In the following, we present a characterization of these automotive vehicular systems.

The development of complex vehicular systems by the automotive maker is a recent trend; they are referred to as "in-car technology" in the automotive business. Some examples of these systems are Toyota Entune, Ford SYNC, MyFord Touch, Renault R-Link, and Nissan NissanConnectSM. These are closed proprietary systems, incompatible with other vehicular systems of other brands and even from different models of the same brand.

Most of these systems provide a little touch screen in the vehicle's dashboard to provide output and input, and their hardware specs are similar to embeded systems of smartphones. Furthermore, these systems can be updated via USB flash drive either at the automotive dealer or by downloading the update to the USB drive. Recently, some automotive companies are starting to provide over-the-air updates to the software of their on-board systems, for example, the Tesla Model S vehicle (http://www.wired.com/autopia/2012/09/tesla-over-the-air).

*3.2. Definition of Context Environment in the Architecture.* The adaptation middleware architecture uses the concept of context to refer to the input information used by the adaptations solutions. From Salber et al. [24], the definition of context refers to the environmental information that is part of a systems operating environment and that can be sensed by the system.

For the adaptation middleware architecture, we define the context of a node, or the context environment of the middleware, as a collection of "context elements." A context element can be provided by an entity inside the node or middleware, called a "context source." In the architecture, we view a context source as an independent module running concurrently and supplying the current value of the context element it offers. At run time, the middleware will take charge of retrieving the current value from the context source either via polling or publish and subscribe mechanism. The architecture defines four context sources: an NL protocol, an application, and finally entities that offer local-node context elements (e.g., information from a GPS or an automobile sensor); each context source could offer multiple context elements.

Each context element has meta information associated to it like a name, the data type of the context element, and so forth. At run time, at a given time $t$, a context element CE has

a value CE($t$), which can be retrieved by querying its context source. At run time, a context element value is encapsulated by the ContextElementValue class, which contains a name field to store the name of the context element, the value field to store the current value of the context element (to be able to store any data type in java, an Object class is used), and a type field to store the data type of the context value (in java a Class class is used).

A context manager module in the middleware handles the retrieval of the current context element value from the source via either the polling or publish and subscribe methods, concurrently to the execution of the adaptation solutions. The PLUGAM architecture is flexible by allowing the context sources to offer the current values only by the polling method, and implement the publish and subscribe method optionally; the context manager favors using the publish and subscribe method if available. The context manager stores the current values of the context elements in an internal direct access memory structure (e.g., hash tables).

The idea is to always have an up-to-date context environment; thus, at the time when an adaptation solution is executed and it needs the latest values of the context elements, the context manager simply gets these values directly from the context manager direct access structure, instead of going to the context source to get the current value. We adopted this context element values scheme to generate a minimal processing time in the execution of an adaptation solution. However, it opens the possibility of getting a not up-to-date value of a context element at the moment when the adaptation solution is being executed.

The processing overhead of the context manager can be reduced by using the publish and subscribe method because the architecture already assumes that the context sources are concurrent entities. In addition, favoring the publish and subscribe method instead of polling avoids getting non-up-to-date values of context elements, because the context source immediately informs the context manager when the value changes.

For critical adaptation solutions that depend on the most updated value of the context elements to work correctly, we can trade up-to-datedness for processing time by extending the adaptation configuration definition. This extension involves adding a Boolean-type flag named GetInmediateValue alongside the context element field in the adaptation configuration. This flag will be used at the moment the adaptation solution is being executed and getting the current values of the context elements; if the GetInmediateValue flag is true, then the middleware does not get the current value from the context manager, but instead goes and retrieves the value by directly querying the context source.

*3.3. Definition of the Adaptation Modes.* As we mentioned earlier, the concept of adaptation mode refers to where or when in the process will an adaptation solution be executed. In order to produce adaptations involving a multiple protocol network layer and adaptations that change the NL protocol internal parameters, in this version of the architecture we define four modes of adaptation; these modes are as follows.

(i) *Send:* an adaptation can take place the moment the message is sent, in between when the application calls the send command and before arriving to the network layer protocol.

(ii) *Receive:* an adaptation can take place the moment the message is received, in between after the NL protocol delivers the message to the upper layer and before the application actually receives it.

(iii) *Forward:* an adaptation can also take place the moment the message is forwarded, meaning when the protocol receives a message from another node to relay it, just before it will resend it to another node.

(iv) *CUPIB:* finally, the architecture supports the CUPIB mode (continuous update of protocol internal behavior) used for adaptation of behavior inside NL protocols by periodically executing the adaptation solution.

To further understand the send, receive, and forward adaptation modes, Figure 3 shows their location in regard to the end-to-end path of a message passing through the application and network layers.

These adaptation modes were deducted from examining prior adaptive protocols proposals: for example, in [6] the authors perform adaptation from the receive network layer operation, in [3, 6] the authors perform adaptation in the forward operation of a network layer protocol, and in [11] the authors perform adaptation in the sending of the messages in the NL protocol. Finally, the work in [8] performs adaptation to NL protocol by updating an internal behavior of the protocol and reevaluating with a certain frequency.

The CUPIB mode is especially useful for extending already implemented protocols where a constant can be tweaked to adapt to some context conditions. By creating an adaptation solution in the CUPIB mode and linking the protocol variable to an output adaptation action, the adaptation middleware will automatically take care of updating the protocol variable at run time by periodically executing the adaptation solution. The adaptation solution is reexecuted at a certain time interval defined in the configuration of a CUBIP mode instance. In addition, thanks to the definition of an adaptation solution, which allows associating multiple adaptation actions as output, it is possible for a single adaptation solution in CUPIB mode to update various NL protocol variables even from different NL protocols.

*3.4. Encapsulation of the Extension Components Using Plug-Ins.* Back in the adaptation model, we separated the concept of adaptation solution into the contexts elements, the adaptation algorithm, and actions. Moreover, the architecture proposes a separation of the implementation of these adaptation parts, in addition to the implementation of the applications and NL protocols; in this manner all of them can be developed separately and independently. We named these independent modules as the extension components of the middleware. For this purpose, we have chosen to encapsulate these extension components into plug-ins. The idea is to develop all the needed extension components inside the plug-ins, and load them into the middleware by loading the plug-ins.

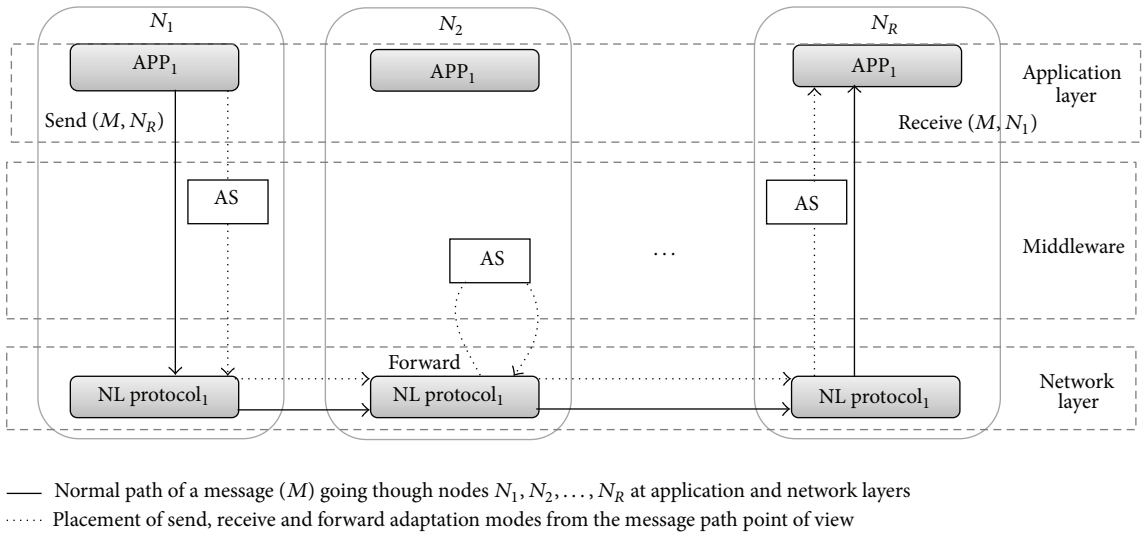


Figure 3: Placement of the send, receive, and forward adaptation modes from the message end-to-end path between the application and network layers.

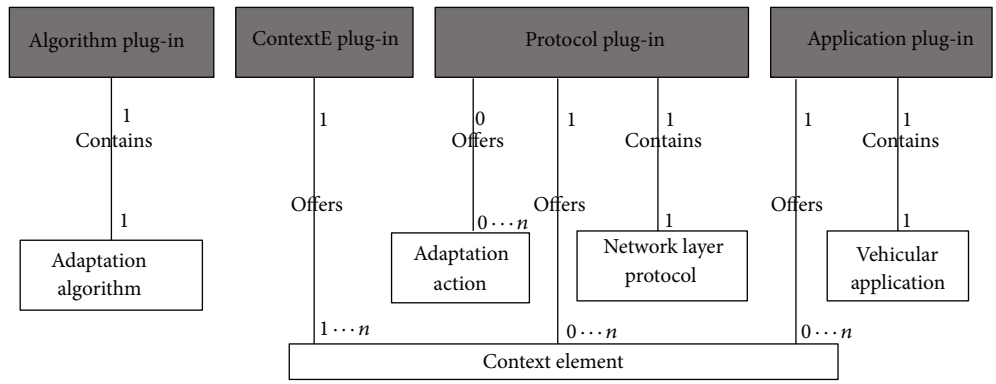


Figure 4: UML ER diagram showing the relationship between the types of plug-in and the extension components.

Plug-in platforms represent a flexible approach for building software systems which are extensible and customizable to the particular needs of the individual user [25]. Plug-in platforms enable the extension of a core application with new features implemented as components that are plugged into the core at load time or even at run time and integrate seamlessly with it. The best known use of plug-in technology is in web browsers and the Eclipse platform (Eclipse webpage: http://www.eclipse.org/). In addition, the popular Unik-olsrd (Unik-olsrd website: http://www.olsr.org/) implementation of OLSR supports a plug-in framework based on encapsulating the plug-ins in dynamic libraries. The core application uses a plug-in by following a service interface. A plug-in is an already compiled application piece that provides a certain, usually very specific, function or service "on demand". The plug-ins depend on the mechanisms provided by the core application and do not usually work by themselves.

Separating the implementation of the extension components and encapsulating them using plug-ins greatly facilitate the sharing of these components created by third party middleware users (or a separate team in the project), thus accelerating the development of new adaptation solutions by reusing extension components. Another benefit is for a plug-in developer to avoid the interaction with the middleware source code when developing the plug-in, even avoiding compiling the whole middleware. Experience tells us that learning to extend and compile an already developed system (possibly of thousands of lines of code and files) is not trivial and is time consuming; hence, this can greatly increase the development time.

The architecture defines four different types of plug-ins, each with their own service interface to offer the middleware the different extension components. In addition, each of these different plug-in types shares a common set of methods giving metainformation about the plug-in that will serve to identify it, such as version, authors names, name of plug-in, and type. The four types of plug-ins and their relationship with the extension elements are shown in Figure 4 these are as follows.

(i) *The application plug-in*, offering only one application component. The application plug-in can also offer

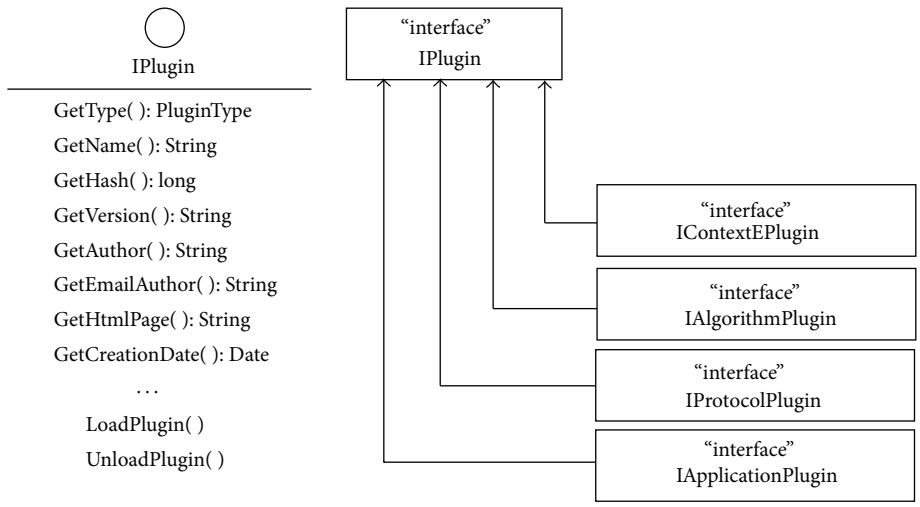


Figure 5: IPlugin interface definition and the relationship with the plug-in types.

multiple context elements to the context environment, since an application is a context source.

(ii) *The contextE plug-in*, which offers multiple context elements components. The aim of this type of plug-in is to offer contexts elements from the node or vehicle, not tied to an NL protocol or application, for example, a contextE plug-in wrapping around a GPS receiver offering time, speed, and location information; another example could be a contextE plug-in offering context elements related to some sensor of the vehicle.

(iii) *The protocol plug-in*, which offers one NL protocol, offers multiple context elements in relation to the NL protocol and multiple adaption actions to change the internal protocol variables.

(iv) *The algorithm plug-in*, which offers one adaptation algorithm.

We finish describing the plug-in aspect of the architecture by showing the plug-in interfaces (APIs) used to construct the plug-ins and offer the extension components to the middleware. The plug-in APIs consist of four interfaces for the four types of plug-ins in the middleware, plus the IPlugin interface. The IPlugin interface contains the set of methods to provide metainformation about the plug-in this interface must be implemented by every type of plug-in, or in object oriented perspective, the other plug-ins interfaces extend from the IPlugin interface (see Figure 5).

Finally, we present the definition of all the different plug-in interfaces of the architecture in the appendix.

*3.5. Identification Mechanism of Extension Components.* As we explained in a previous section, the extension components of the architecture are created independently of the middleware. The architecture needs a method to uniquely identify each extension component and refer to them at load time and run time, even between those from the same type. In this section, we describe how we address this issue.

We defined two ways to uniquely identify an extension component depending on a one-to-one or one-to-many relationship between the plug-in and the extension component. For the extension components having a one-to-one relationship with the plug-in (NL protocol, the application, and the adaptation algorithm), these extension components are uniquely identified from all the others by taking into account the duple (NP, HP), where:

(i) NP is the name of the plug-in containing the extension component and also the name of the extension component;

(ii) HP is the hash value of the plug-in containing the extension component.

For extension components having a one-to-many relationship between the plug-ins, like the context elements and the adaptation actions, these extension components are uniquely identified from all the others by taking into account the tuple (NP, HP, and NE), where:

(i) NP is the name of the plug-in containing the extension components;

(ii) HP is the hash value of the plug-in containing the extension component;

(iii) NE is the name of the extension component. We assume that this value is unique across the other extension components contained in the same plug-in.

To store and represent the identification information of the extension components, we proposed the use of the ElementIdentifier class shown in Table 5(a). ElementIdentifier is simply a container class with three data fields that will contain the identity information needed by the different types of extension components of the architecture. The use and meaning of these three data fields are shown in Table 5(b).

*3.6. The Adaptation Configuration.* After all the needed extension components are loaded in the middleware, the next

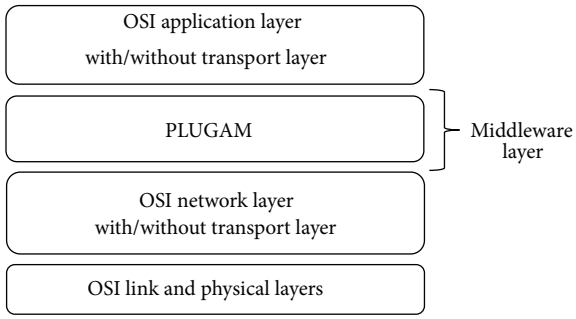


Figure 6: Adaptation middleware position in the OSI model.

step is to build the adaptation solution by expressing their relationship. The mechanism to describe an adaptation is by declaring an adaptation configuration (see Table 6); here we state the associated ElementIdentifier references of the adaptation algorithm we want to use, what context elements are needed as input, and the adaptation actions associated to the output values. In addition, we state the adaptation mode where this adaptation solution will be executed and assign a name for the configuration. An adaptation configuration also contains other additional fields that are adaptation mode specific; these are as follows.

(i) *Protocol and application fields*: these fields are needed by an adaptation configuration in the send, receive, and forward adaptation modes. Its purpose is to tie a configuration to a network operation between a specific application and protocol; that is, a send configuration associated when application $X$ calls the send command to NL protocol $Y$.

(ii) *The CUPIBTime field*: this field is needed only by adaptation configurations in CUPIB mode and it is used to specify the number of milliseconds needed to reexecute the adaptation solution.

The architecture allows the middleware user to create any number of adaptation configurations, thus enabling the definition of multiple adaptation solutions in an adaptive network layer.

*3.7. Middleware Aspect of the Architecture.* The PLUGAM architecture is conceived as a middleware system, situated between the application and network layers because the interception of messages between the applications and NL protocols is needed to enable the send, receive, and forward adaptation modes. The middleware aspect also serves to help manage the multiple NL protocols and multiple applications aspect of the architecture. There is no direct communication between applications and the NL protocols; instead, they communicate indirectly using the services offered by the middleware. In reference to the role of a possible transport layer in the architecture, the services of this layer will need to be embedded in either the NL protocol or the application. Figure 6 shows the position of the middleware from the point of view of the OSI layers.

The multiple NL protocol aspect of the middleware architecture facilitates the use of different communication channels (WIFI, 802.11p, 3G, GSM, etc.) in the system. This can be achieved by encapsulating each communication channel lower level protocols (physical and link layer) plus the NL protocol in a protocol plug-in. If two NL protocols in the middleware share the same communication channel (e.g., one for unicast communication and another for dissemination, each sharing the 802.11p radio), possible interprotocol interference must be dealt with by the middleware user. Furthermore, having too many protocols is not desirable, because they are complex modules and may quickly bloat the middleware; finding an optimum set of NL protocols that serve all the applications communications needs is out of the scope of this work.

*3.8. Special Adaptation Actions of the Architecture.* As we mentioned in the adaptation model, the NL protocols offer different adaptation actions to allow the modification of its behavior. In addition, this definition gets extended to account for special adaptation actions, we called them special because they are offered by the middleware and not the NL protocols. The special adaptation actions are used to allow adaptations where the result is a change of behavior between NL protocols and applications. Based on the foregoing and exploring some possible modifications in the interactions between NL protocols and applications, we extend the definition of adaptation actions to conceive three types of adaptation actions in the architecture:

(i) protocol adaptation actions,

(ii) special switcher adaptation action,

(iii) special filter adaptation action.

The protocol adaptation actions are offered by NL protocols and their purpose is to change the behavior of the protocol by linking it to a protocol internal variable or parameter. An NL protocol can have multiple adaptation actions, each linking to an internal variable. For example, a dissemination NL protocol can offer an adaptation action linked to extending or lowering the dissemination spread via changing the value of an internal variable of the protocol, or an adaptation action linking to the HELLO interval constant of the OLSR NL protocol.

The purpose of the special switcher adaptation action is to redirect a message from the predefined path between NL protocols and applications; this adaptation action is used for doing a change between NL protocols adaptation. The switcher receives as input an ElementIdentifier object with the reference to the protocol or application to redirect the message to. The actual decision of where to redirect the message is done in the adaptation algorithm, then by linking to one of the adaptation algorithm outputs the switcher receives the final decision and redirects the message. If the switcher is included in an adaptation solution located in the send and forward adaptation modes, the switcher redirects between the available NL protocols that the application registered at the initiation step. If the switcher is included in an adaptation solution located in the receive adaptation

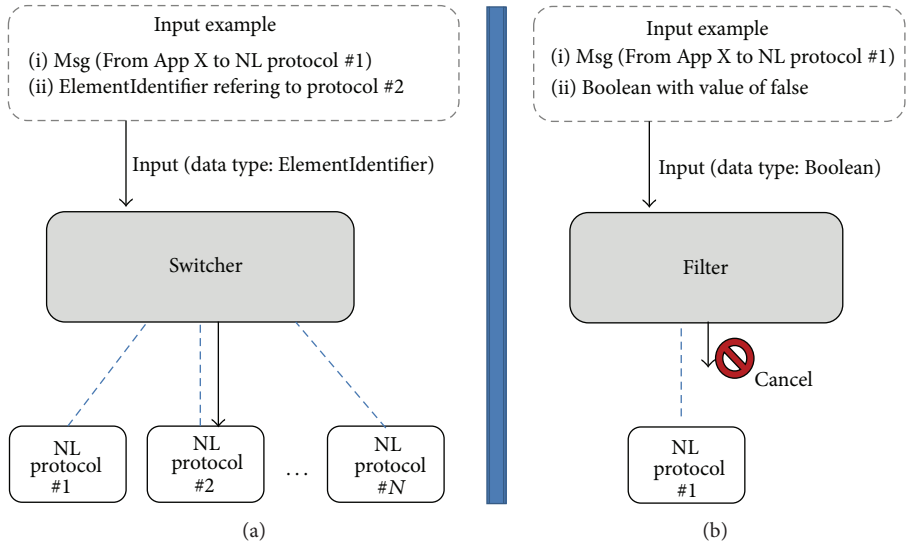


Figure 7: Graphical representation of the special adaptation actions switcher and filter.

mode, the switcher redirects between the applications having registered the NL protocol where the message comes from.

Figure 7(a) shows an example of how the switcher works. In this example, the adaptation solution having the switcher is located in the send adaptation mode, and the default path of the message was to go the NL protocol no. 1, but the ElementIdentifier reference received as input told the switcher to redirect the message to NL protocol no. 2.

The purpose of the special filter adaptation action is to recreate a pass or not to pass behavior. This adaptation action can only be used in the send, receive, and forward adaptation modes. The filter receives as input a Boolean data type where a value of true means that the filter lets the message pass to the destination protocol or application; with a value of false the message is dropped. Figure 7(b) shows an example of how the filter works; in this example the adaptation solution having the filter receives the value of false, the filter then drops the message, and it does not go to the NL protocol no. 1.

To finalize the description of the adaptation actions, the architecture introduces the AdaptationActionValue class to represent the value of an adaptation action at run time similar to the context element value concept. An adaptation action value can also be of any data type similar to a context element value, which is why the definition of the AdaptationActionValue class contains exactly the same attributes as the ContextElementValue class.

*3.9. Interaction of Applications in the Adaptation Middleware.* In this section, we describe how a vehicular application interacts with the middleware and associates to the NL protocols that it will use during its execution. Foremost, the middleware has control of the lifetime of the applications. Applications are started by the middleware and stopped when the middleware terminates.

Furthermore, an application will not interact directly with the NL protocols, instead it uses the middleware as a proxy.

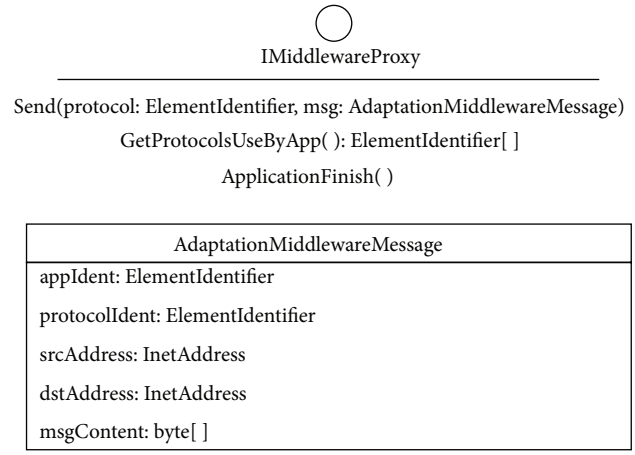


Figure 8: Definition of the AdaptationMiddlewareMessage class and the IMiddlewareProxy interface.

Two-way communication between applications and middleware is achieved using two interfaces: the IApplicationPlugin and the IMiddlewareProxy. The middleware communicates to the application by calling methods of the IApplicationPlugin interface (defined in the appendix in Figure 20). An application communicates with the middleware by calling methods in the IProxyMiddleware object; the definition of the IProxyMiddleware interface is shown in Figure 8.

When the middleware registers the available applications, the middleware creates a middleware proxy object tied to each registered application, then the middleware passes this object to the respective application using the SetMiddlewareProxy method of the IApplicationPlugin interface.

Moreover, at the application's registration step the middleware queries the application via the GetAppDeclaredProtocol method of the IApplicationPlugin interface, this

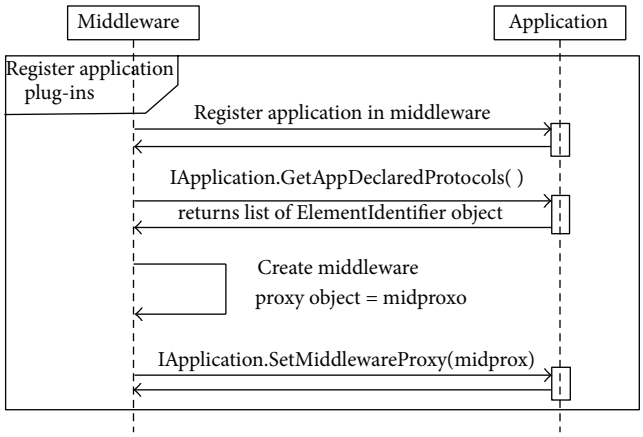


Figure 9: UML sequence diagram of the registration of an application by the middleware.

method returns a list of references (ElementIdentifier objects) of all the NL protocols that the application needs. The middleware must search if these NL protocols where indeed loaded in a prior step. After the application starts, it must use the proxy middleware object to interact with the requested NL protocols. In Figure 9, we show the sequence diagram of the steps done by the middleware to register an application, here is where the middleware registers the requested NL protocols of the application and passes the middleware proxy object to it.

At run time, when an application wants to send a message to an NL protocol, it must call the send method of its middleware proxy object. This send method receives as input an ElementIdentifier object with a reference to the NL protocol chosen to send the message, in addition to an AdaptationMiddlewareMessage object wrapping the contents (an array of bytes) of the message to send; the definition of the AdaptationMiddlewareMessage class is shown in Figure 8. Moreover, the middleware calls the onReceiveMessage method of the IApplicationPlugin to transfer a message receive from an NL protocol to the application.

The dstAddress field in the AdaptationMiddlewareMessage object must be filled by the application before sending the message, in order to instruct the NL protocol and the destination node of the message. In this version of the architecture, we do not address the issue of having different representations of a network layer address depending upon the different types of NL protocols; for instance, for a dissemination or geocast protocol the destination address could be difficult to be represented by an IP address. We will leave this issue as future work and for now assume that using an IP address will suffice.

The appIdent and protoIdent fields of the AdaptationMiddlewareMessage contain the references of the application and NL protocol that originally sent the message. These fields are assigned by the middleware after the application calls the send method in the middleware proxy, the application can view this information when it receives the message. The appIdent field is specially useful when the middleware redirects the message to the right application; this is analogous to the TCP port of the TCP/IP stack. Finally, the scrAddress field of the AdaptationMiddlewareMessage is assigned by the NL protocol when sending the message, and when receiving a message this field can be consulted by the application.

## 4. Description of Case Study Examples

In this section, we present two simple case studies of adaptive protocols built with our adaptation middleware architecture, each presenting two distinct domains of the middleware behavior. These case studies will help us better explain how all the aspects of the architecture work together in order to build an adaptive network layer and also as a proof of concept of the architecture. The first case study is about extending an existing MANET NL protocol to be more dynamic by adapting it based on local context, thus making it better suited to be used in VANETs. In the second case study, we propose to implement a conceptual scenario with our architecture, where a selection or switch must be made between two NL protocols. In the following, we present a description of these case studies and how they are implemented with our architecture.

*4.1. Case Study 1: Modifying an Existing NL Protocol to Make It More Dynamic by Adapting to Local Context.* As we previously mentioned, this case study extends an already defined NL protocol by adding the functionality of adapting to certain context information. As the base NL protocol for the case study, we chose one from MANET because there are still no reference protocols for VANET. We selected OSLR because it is particularly well known, with a lot of work around it and with code availability. OLSR (http://tools.ietf.org/html/rfc3626) is a proactive protocol which maintains an up-to-date routing table using periodic messages (e.g., HELLO messages) received from neighbors. Huang et al. [26] experimented with the HELLO_INTERVAL constant which is the time period to send these HELLO messages and it is strongly related to the up-to-dateness of the routing table. They tested performance changing this constant using different scenarios of node density and node mobility. They concluded that adjusting this constant can bring improvements in performance and proposed as future work to adapt it dynamically at run time. We used Huang et al. results and future work as the basis for the adaptation proposal of this case study, this way we have a certainty that the impact of the proposed adaptation will be indeed beneficial.

Associating the Case Study 1 with the concepts of our architecture for adaptation, the HELLO_INTERVAL constant is represented as an adaptation action offered by a protocol plug-in implementing OLSR. The context information which will be the input of this adaptation will be a simplification of the node mobility, the node, or vehicle speed. There are more complex and precise ways to represent the node mobility (e.g., in [21] the mobility metric is based on link duration of the neighbor nodes); nonetheless, to simplify the description of this case study example we think that using the vehicle speed

Table 1: Table showing the $(v_i, f(v_i))$ pairs proposed for the adaptation algorithm of case study 1.

| $v_i$ | $f(v_i)$ |
|---|---|
| 0 | 3500 |
| 10 | 3400 |
| 20 | 3000 |
| 30 | 2000 |
| 70 | 1000 |
| 90 | 900 |
| 120 | 800 |

serves the purpose. We proposed to obtain the vehicle speed by creating a ContextE plug-in wrapping a GPS code which calculates this information by measuring the Doppler shift in the signals from the GPS satellites, this ContextE plug-in will offer the middleware the context element of "vehicle speed."

In addition, we need to describe the adaptation algorithm that we will use for the Case Study 1 adaptation solution. We decided to create an extremely simple adaptation algorithm based on one-on-one mapping between the vehicle speed and the HELLO_INTERVAL value. The adopted vehicle speed and HELLO_INTERVAL relationship is based on the following simple reasoning: at higher vehicle speed the neighbors change more rapidly; hence, the neighbor table must be updated more often by lowering the value of the HELLO_INTERVAL. This reasoning does not take into account the existence of neighbor nodes driving at high speed alongside the vehicle; however, for this example it is useful enough. To implement the one-on-one mapping, we defined the pairs $(v_i, f(v_i))$, where $i \in \aleph$ and $\in [1, N]$, $v_i$ represents the vehicle speed, and $f(v_i)$ represents the future value of the HELLO_INTERVAL constant in milliseconds (2000 milliseconds is the default value in OLSR). The $N = 7$ pairs that we selected are shown in Table 1.

With the help of the $(v_i, f(v_i))$ pairs, we can compute the HELLO_INTERVAL $f(v)$, where $v \in \Re$, as follows:

$$\begin{aligned} f(v) = \text{Linear Interpolation } & \left(v_i, v_{i+1}\right) \mid v \in \left[v_i, v_{i+1}\right), \\ & i \in [1, N-1] \\ \text{if } v < v_1 \text{ then } & f(v) = f\left(v_1\right), \\ \text{if } v > v_N \text{ then } & f(v) = f\left(v_N\right). \end{aligned} \tag{2}$$

Finally, the adaptation solution for Case Study 1 will be located in the CUPIB adaptation mode. This mode will periodically reexecute the adaptation, solution thus updating the HELLO_INTERVAL value; we are defining the CUPIB period (in the CUPIBTime field of the adaptation configuration) to 1000 ms because it is the update rate of a common GPS device.

We now proceed to explain how to build the Case Study 1 in our architecture. The first step is to construct the following plug-ins (see Figure 10).

(i) Create a protocol plug-in of OLSR; this plug-in will offer an adaptation action tied to the HELLO_INTERVAL constant of OLSR.

(ii) We need a ContextE plug-in offering the vehicle speed context element; we decided to obtain it by wrapping the ContextE plug-in around a GPS receiver code.

(iii) Create an adaptation algorithm plug-in that computes the previously described $f(v)$; this algorithm receives as input a Double data type containing the node speed $v$, and outputs a long data type with the resulting HELLO_INTERVAL value $f(v)$.

Once we have all the required extension components, a second step is to link them all by declaring an adaptation configuration as shown in Figure 11; the configuration shown here is in JSON format as we implemented it in the proof of concept prototype. Looking at this JSON code, we can see that most of the elements constituting an adaptation configuration are in the form ElementIdentifier objects represented in JSON as the {"name":, "hash":, "secondname":}. In addition, this adaptation configuration is declared in the CUPIB adaptation mode and the CUPIB_time field is to 1000 ms.

The preceding are all the steps necessary to create the adaptation of Case Study 1 in the adaptation middleware; at run time the middleware will execute this adaptation solution periodically due to how the CUPIB mode works, hence changing the HELLO_INTERVAL constant of OLSR depending on the node speed.

*4.2. Case Study 2: Adaptation Based on Switching between Different NL Protocols.* In this case study, we present an adaptation in which a selection or switch between different NL protocols must be made depending on the current context. To achieve this, we describe a simple scenario where such switch between NL protocols is needed. This scenario is the dissemination of an accident event in the road to interested vehicles. We chose to use a directional broadcast NL protocol to only inform vehicles driving in the same direction as the vehicle which had the accident. This works well when there are enough vehicles driving in the same direction (scenario A in Figure 12). However, if there are not enough vehicles driving in the same direction to relay the message, the communication of the message fails.

We propose to exploit the vehicles driving in the opposite direction, and switch to use a carry and forward protocol given this context (scenario B in Figure 12). The conditions to switch or select the NL protocol are: if there are not neighbors in same direction and we detect a vehicle passing in the opposite direction. If these conditions are met, then switch and use carry and forward protocol; if not continue using the directional broadcast.

For this scenario, we could choose to put the adaptation solution in two adaptation modes: send or forward. Depending on the chosen adaptation mode the behavior is different. If we choose to insert this adaptation solution at the send adaption mode, then the switch between protocol can only happen in the source node, specifically when the application calls the send command to an NL protocol throughout the middleware. If we want to execute the adaptation solution at each hop, we must insert the adaptation solution in the forward adaptation mode. Is it possible to put the same adaptation solution in both the send and forward modes by

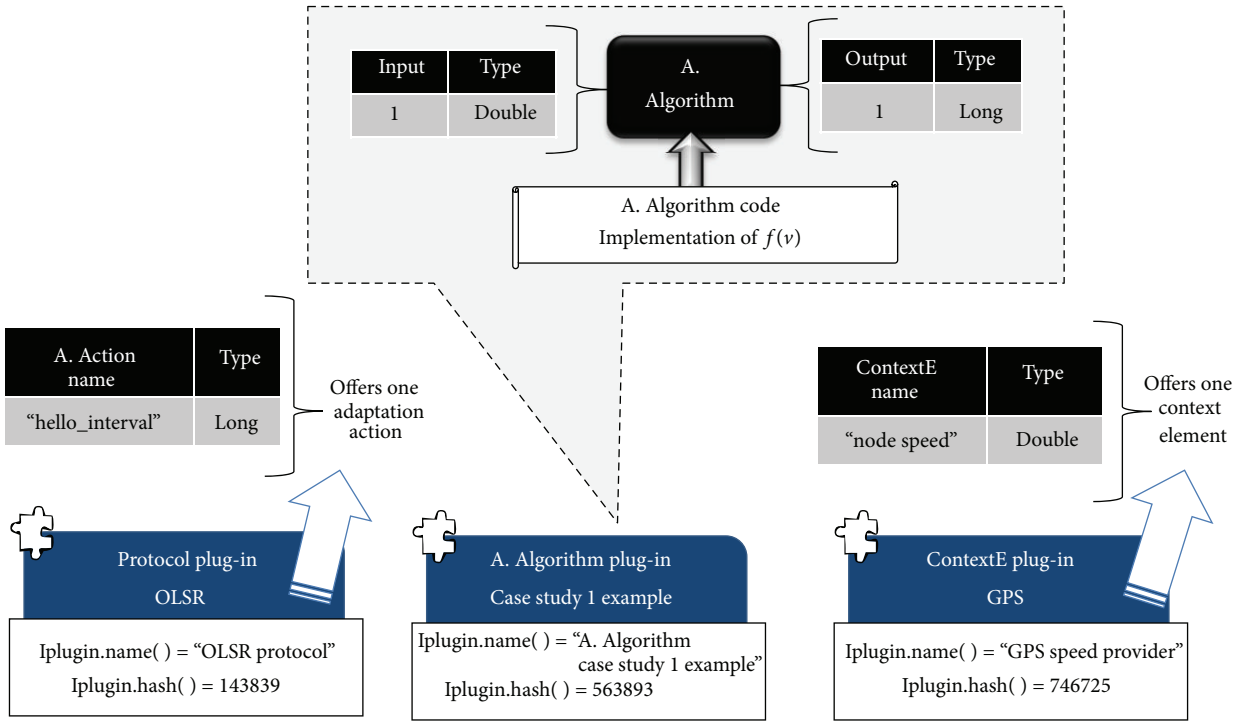


Figure 10: Implementation of the extension components needed for Case Study 1 in the adaptation middleware.

Declaration of the adaptation configuration in JSON format

```
[{
    "configurationName": "Case study 1 configuration",
    "algorithm": {"hash": 563893, "name": "A. Algorithm case study 1 example", "secondName": ""},
    "contextElements": [{"hash": 746725, "name": "GPS speed provider", "secondName": "node speed"}],
    "actionElements": [{"hash": 143839, "name": "OLSR protocol", "secondName": "hello_interval"}]
    "adaptationMode": "CUPIB",
    "CUPIBTime": 1000,
    "protocol": {"hash": 143839, "name": "OLSR protocol", "secondName": ""},
    "application": {"hash": 0, "name": "" , "secondName": ""},
}]
```

Figure 11: Adaptation configuration of Case Study 1 in JSON format.

creating two different adaptation configurations. In Figure 12 we opted to use the send adaptation mode for demonstration purposes.

Having described the Case Study 2 scenario we proceed to show how to build it in our architecture. Foremost is to construct the necessary plug-ins to provide the extension components needed for this adaptation configuration, which is why we constructed the following plug-ins (see Figure 13).

(i) Two protocol plug-ins, one implementing the directional broadcast protocol and another for the carry and forward protocol. To implement the context elements needed for this adaptation solution, for this example, we decided that the carry and forward protocol will provide them. These context elements are very simple Boolean values indicating if there is a neighbor passing by in the opposite direction or not, and if there are neighbor vehicles driving in the same direction or not. Nevertheless, computing the value of these context element is not trivial, the implementation of how to compute them is up to the protocol implementation and not shown in this example.

(ii) We need an application plug-in with the accident in the road application; this application requires the directional dissemination and the carry and forward NL protocol to work; hence, their ElementIdentifier objects must be included in the list returned by the GetAppDeclaredProtocols method of the IApplicationPlugin interface.

(iii) Finally, we need an adaptation algorithm plug-in containing the logic of how to select the protocol to

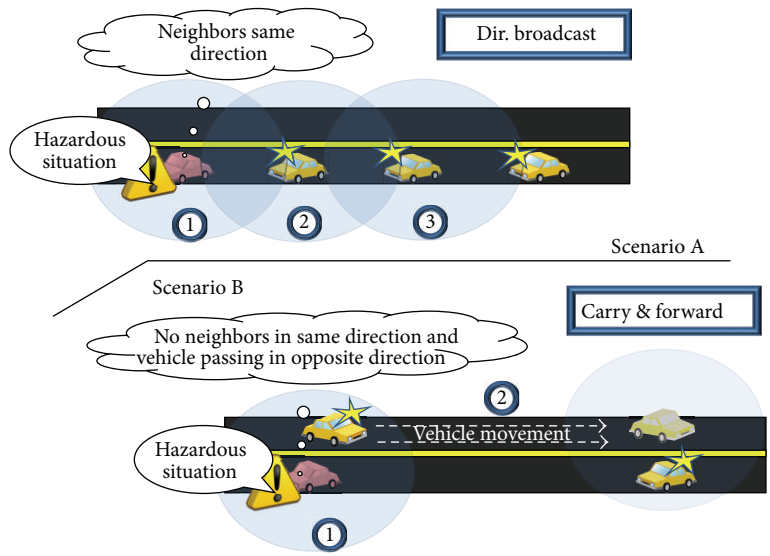


Figure 12: Adaptation configuration of Case Study 1 in JSON format.

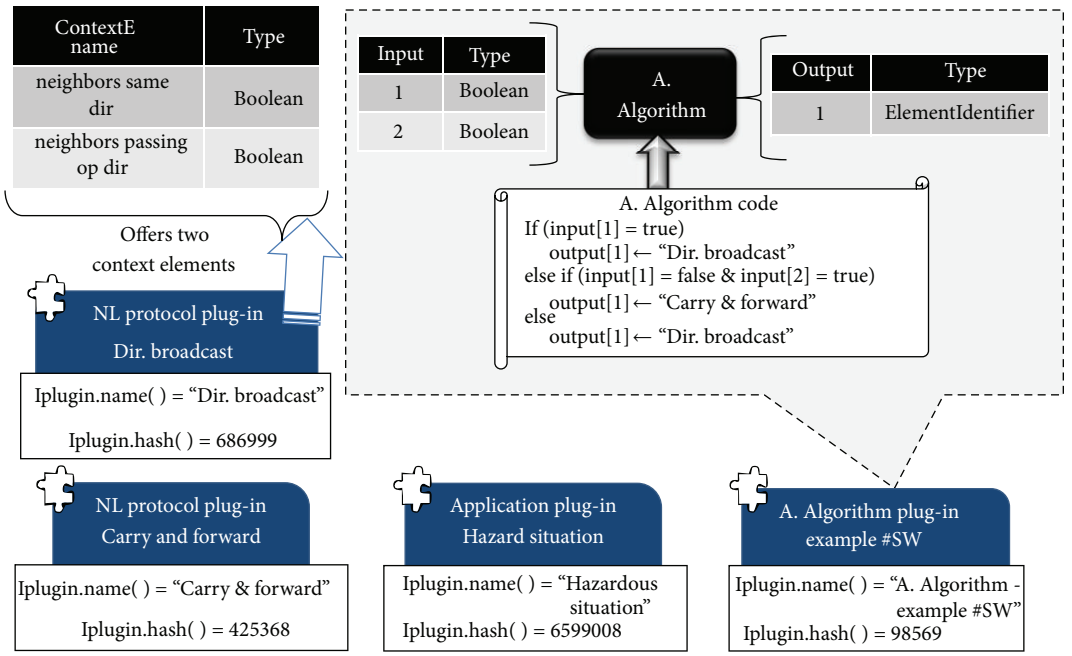


Figure 13: Implementation of the extension components needed for Case Study 2 in the adaptation middleware.

switch. The output type of this adaptation algorithm must match the parameter type of the special adaptation action switcher, which is an ElementIdentifier data type that refers to an NL protocol. The algorithm contains a very simple conditional operation using the two Boolean context elements.

Next, we need to link the functionality of the extension components by declaring the adaptation configuration, Figure 14 shows the adaptation configuration of Case Study 2 expressed again in JSON format. An important remark about an adaptation configuration created for the send adaptation mode, is that the application and protocol fields must also be assigned in order for the adaptation to be executed only when the hazardous situation application sends a message to the directional broadcast protocol (which is the default NL protocol for this scenario).

Once the middleware loads the adaptation configuration, it will execute this adaptation at run time. To finish the description of this case study example, we will present how this adaptation is performed inside the middleware at run time (see Figure 15). It all starts when the hazardous situation application calls the send command to disseminate the message using a directional broadcast protocol (1). Instead

Declaration of the adaptation configuration in JSON format

```
[{
  "configurationName": "Adaptive example #SW",
  "algorithm": {"hash": 98569, "name": "A. Algorithm - example #SW", "secondName": ""},
  "contextElements": [{"hash": 425368, "name": "Carry & forward", "secondName": "neighbors same dir"},
                      {"hash": 425368, "name": "Carry & forward", "secondName": "neighbors passing op dir"}],
  "actionElements": [{"hash": 0, "name": "#SPECIAL#", "secondName": "#SpecialASwitcher#"}]
  "adaptationMode": "Send",
  "CUPIBTime": 0,
  "protocol": {"hash": 686999, "name": "Dir. broadcast", "secondName": ""},
  "application": {"hash": 6599008, "name": "Hazardous situation", "secondName": ""},
 }]
```

Figure 14: Adaptation configuration of Case Study 2 in JSON format.

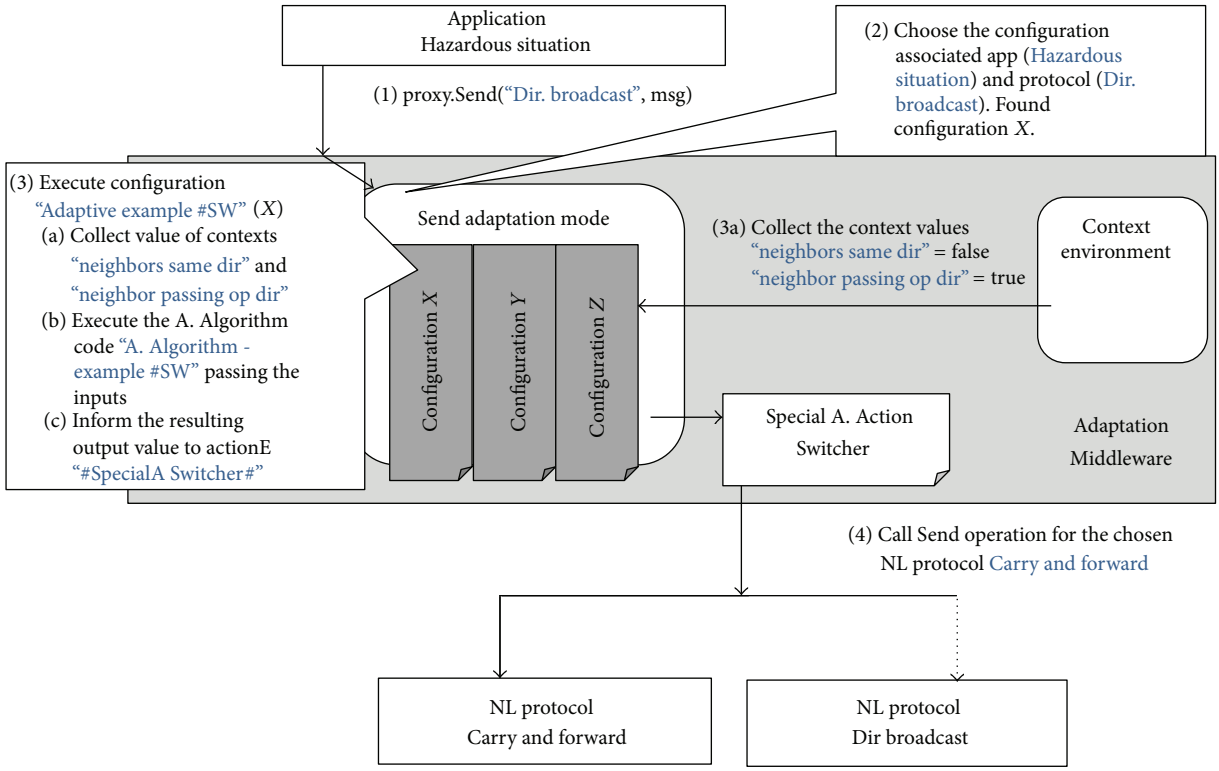


Figure 15: Graphical description of the run time behavior of the middleware executing the adaptation configuration of Case Study 2.

of going directly to the protocol, it is intercepted by the middleware and goes to the send mode module. Here, a lookup is performed to see if there is a configuration defined at initiation for the application-protocol pair (2), which indeed there is and it's named "Adaptive example #SW". Next is the execution of the adaptation solution declared in the configuration (3); first the middleware gets the current context element values from the context environment via the context manager (3.a); then these values are passed as inputs to the adaptation algorithm which executes and produces the output value needed to the adaptation action, in this case the special adaptation action switcher. The middleware then transfers the value to the switcher module which does the actual redirection of the message to the chosen NL protocol (4). In the run time example of Figure 15, the context elements are such that the message is directed to the carry and forward protocol.

## 5. Description of Proof of Concept Prototype

As proof of concept, we decided to implement a prototype of the PLUGAM adaptation middleware. An important aspect for the prototype was to find a proper plug-in platform. In addition, working on an implementation of the PLUGAM

architecture helped us better define some lower level details of the architecture and have a more solid proposal. In this section, we present the implementation details of this prototype.

In regard to the search for a plug-in platform, we found some plug-in frameworks, platforms in Java and C#, and HTML programming languages. For example, the Chrome extension framework allows the creation of extensions (plug-ins) developed in HTML and packed inside a zip file containing the HTML code and a manifest in JSON format to store the extension's metadata. In the case of more traditional programming languages like C# and Java, the binary code of the plug-in can be packed as a DLL file in C# .NET [25] and as a JAR file in Java. To identify that a binary file is a plug-in instance and to manage plug-in metadata, .NET attributes and reflection services can embed metadata directly in the DLL file; Java relies on reading metainformation defined in a manifest text file inside the JAR file.

We selected OSGi and iPOJO as the base technologies for the plug-in platform. We chose the OSGi platform (OSGi platform specification homepage: http://www.osgi.org/Technology/WhatIsOSGi) which is developed using the Java programming language. OSGi defines an architecture for modular application development, these modules are JAR files which in OSGi are named bundles. A plug-in from our architecture is a bundle that offers services available in the OSGi environment, and the adaptation middleware will be another bundle accessing the plug-ins services. The main reason to select OSGi as the plug-in platform is because the OSGi is a solid platform; also its community is currently very active, there is a lot of development to improve and extend the platform and there are a lot of additional tools to deal with bundles.

The other technology constituting the plug-in platform is iPOJO (iPOJO webpage: http://felix.apache.org/site/apache-felix-ipojo.html) and its purpose is to simplify OSGi application development and promotes service-oriented programming on OSGi platforms. Based on the concept of POJO (Plain Old Java Object), application logic is developed easily. A PLUGAM plug-in will be implemented as an iPOJO service. Thanks to iPOJO, there is no need to make reference or calls to OSGi code to be able to implement an application module via OSGi, the linking of OSGi and the iPOJO component service to the application logic is made at the compilation level. In other words, the use of iPOJO reduces the development of a plug-in in the prototype to simply writing a normal plain Java class that implements a Java interface, this interface being one of the PLUGAM plug-in interfaces.

The OSGi platform specification has many different open source implementations commonly known as OSGi containers, for example, Equinox OSGi, Apache Felix and Knopflerfish OSGi. In our case, we selected Apache Felix (Apache Felix webpage: http://felix.apache.org/site/index.html) as the OSGi container because its development and community are very active, and it is used as a base for industry quality architectures like ServiceMix (Apache ServiceMix is an open source Enterprise Service Bus: http://servicemix.apache.org/home.html).

We implemented the adaptation middleware as an OSGi bundle that will be running in the same OSGi container as PLUGAM plug-ins. When the middleware bundle is started by the user, it will detect all the plug-ins which are already loaded in the OSGi container. In addition to developing the plug-in platform of the prototype, we implemented some test plug-ins. Specifically we implemented the necessary plug-ins to implement the Case Study 1 example in the prototype.

(i) An OLSR protocol plug-in which is developed using the OLSR code found inside the Mchannel [27] middleware implementation. This plug-in offers an adaptation action to modify the HELLO_INTERVAL of OLSR.

(ii) A contextE plug-in offering GPS context information by connecting to a Bluetooth GPS; we modified the base code of the GPSylon project (GPSylon webpage: http://www.tegmento.org/gpsylon/) to create the plug-in. This plug-in offers two node speed context elements, one obtained from GPS NMEA readings and the other one calculated from two positions; it also offers longitude, latitude, time, and number of satellites caught by the GPS.

(iii) An adaptation algorithm plug-in of the simple vehicle speed to HELLO_INTERVAL mapping we talked about in Section 4.1.

Thanks to the use of iPOJO creating a plug-in for the prototype is rather simple and involves doing the following steps using the Eclipse development tool.

(i) Create a new Java project in Eclipse, then import the PluginServices.jar in your project; this jar is also an OSGi bundle which contains all the auxiliary classes and the plug-in interfaces for the middleware and plug-ins to interact with each other.

(ii) Create a class in the project and make it implement one of the plug-in interfaces.

(iii) Compile the project into an iPOJO bundle using Apache Ant and the input files build.xml, metadata.xml, and ⟨Project name⟩.bnd; these files must be customized for each plug-in project.

(iv) Put the resulted bundle in the plug-ins folder of Apache Felix.

Installing the prototype is also simple thanks to Java and Apache Felix, involving only to copy the contents of the Apache Felix folder with the bundles of the adaptation middleware to the computer, then put the self-created plug-ins in the predefined folder of Apache Felix.

To run the middleware prototype, we must start the Apache Felix JAR file (as shown in (1) of Figure 16), wait a few seconds until Apache Felix detects the installed plug-ins, then in the command console of Apache Felix start the bundle of the PLUGAM adaptation middleware (as shown in (3) of Figure 16).

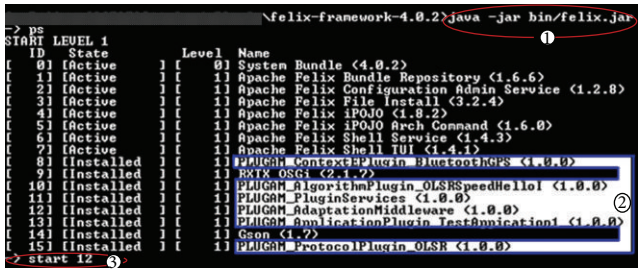


Figure 16: Apache Felix running with the adaptation middleware bundle, the pluginServices bundle, and the PLUGAM plug-ins of Case Study 1 which are shown in blue (2).

## 6. PLUGAM Evaluation

The evaluation of the adaptation middleware is presented in two phases. The First phase is a cost/performance evaluation, which is based on a set of measurements to the prototype in order to get a rough idea on the cost of using the middleware in the vehicular system. The objective of these measurements is to determine if such middleware architecture is viable to implement in a future vehicular system. Moreover, in addition to presenting two case studies examples as proof of concept of the proposed architecture, in the second phase of the evaluation we go further and perform a set of experiments/simulations to analyze the impact of one of these specific adaptations, the Case Study 1 example, in order to confirm if in practice this adaptation solution indeed brings performance gains as proposed in theory by Huang et al. [26]. Finally, we discuss the scalability of the proposed architecture and talk about the additional costs introduced by our generic approach in contrast with an existing adaptive protocol, which is a particular solution.

*6.1. Performance/Cost Measurements.* To evaluate the performance or cost of the adaptation middleware, we designed a set of measurements to be done to the prototype. The aim of these measurements is to analyze the possible processing and memory cost of using a plug-in platform and the processing cost added by the middleware between the interactions of the applications and NL protocols. The objective of this evaluation is to determine if such middleware architecture is viable to implement in a future vehicular system.

For the performance/cost evaluation, we made measurements to the following aspects of the prototype.

(i) Memory requirements of the prototype.

(ii) Processing costs or additional latency generated by the adaptation middleware when executing adaptation solutions in all the adaptation modes.

(iii) Finally, we are interested in analyzing the memory and processing overhead introduced by the prototype's plug-in platform.

All the measurements performed to the prototype were made on a MacBook Air model A1231 (2008), which contains an Intel Core 2 Duo 1.6 GHz and 2 Gb of RAM. On the software side, we used the Ubuntu 10.10 operating system, the installed software is Java JRE 1.6.0.22, Apache Felix version 4.0.2, and iPOJO 1.8.2. We believe that these hardware and software specifications will be similar to the onboard systems of two-three years future vehicles, with a price point of $700–1100, using computer technology from four or five years behind and most probably using a version of Linux or Windows Embedded. For example, the MyFord Touch costs around $1150 and uses Microsoft Windows Embedded Auto software platform; it has an ARM Cortex A8 600 MHz CPU with 512 MB RAM and 2 GB NAND flash memory. The Toyota Entune costs around $1050 (http://toyota-entune.com/toyota-entune-cost/). On another aspect, all the measurements of the prototype were done a hundred times and we are showing the average value and confidence intervals.

*6.1.1. Prototype Memory Requirement.* Due to the possibility and interest of deploying these vehicular systems in embedded systems or even smart phones, which have limited processing power and specially memory limitations, it is of interest to have an idea of the memory costs of using the adaptation middleware, more specifically the ROM and RAM usage. The general memory requirements obtained from the prototype are: 20.1 MB of RAM and 111.9 MB ROM memory (size for PLUGAM binary files).

The above RAM value was obtained from the memory footprint of the java process reported by the process status (ps) program in Linux. A deeper examination of the 20 MB RAM usage of the prototype reveals that 16.17 MB are needed to run the plug-in platform environment (iPOJO + Apache Felix + JVM), the JVM alone consumes 6.72 MB. On the other hand, from the previous ROM value, 109.5 MB correspond to the JVM files, only 1.98 MB correspond to the Apache Felix with iPOJO files, and the last 442 KB correspond to the adaptation middleware files. Moreover, if we include the plug-ins needed to implement the Case Study 1 example in the prototype, the ROM increases an additional 4.03 MB.

Furthermore, an important memory overhead of the adaptation middleware is the addition of a middleware specific header to the messages generated by the applications. This header contains the information of the AdaptationMiddlewareMessage object (shown in Figure 8). Since the fields of the AdaptationMiddlewareMessage has ElementIndentifier objects which contains strings, the size of the header depends on the maximum number of characters for these strings. Setting this maximum size to 30 characters of one byte each, and using an IPv4 address representation of 4 bytes resulted in a header size of 132 bytes.

In summary, the resulting ROM and RAM requirements are modest and should fit even on today's low cost Android smartphones, which generally have 512 MB of ROM and RAM; we highlight Android devices because Apache Felix can be configured to run on Android systems, thus making it feasible to run the adaptation middleware in them. The plug-in platform implementation using Apache Felix is the source of most of the RAM usage of the PLUGAM prototype. The JVM is responsible for most of the ROM usage of the prototype. Finally, the resulting header's size is high

and could be reduced by redefining the ElementIdentifier class, preferentially by omitting the use of strings in the identification mechanism of the architecture.

*6.1.2. Prototype Processing Overhead.* The goal of obtaining these measurements is to have an approximation about the processing cost (latency) induced by using the adaptation middleware. In the send, receive, and forward adaptation modes, the middleware intercepts the passing of the messages between the vehicular applications and the NL protocols and executes an adaptation solution in between. In the case of the CUPIB mode, the recalculation of the variables inside the NL protocols also depends on how much time the adaptation solution is executed in this mode. We are interested in knowing the processing time of executing an adaptation solution with the different variations of adaptation modes, with the special adaptation actions, and with different types of adaptation algorithms.

Because the instructions and operations inside an adaptation algorithm are defined by the plug-in developer, inside there could be any number of instructions and complex operations. For the purpose of obtaining measurements with different types of adaptation algorithms, we defined four profiles of an adaptation algorithm, we named them "algorithm load" and they are based on quantity of operations with floating point numbers, the devised algorithms loads are as follows.

(i) *None:* an adaptation algorithm without any instructions or operations.

(ii) *Low:* represented by an $O(n)$ algorithm and implemented by an average estimation algorithm of a list of $n$ real numbers. This load is intended to represent an algorithm that contains some variable assignments, comparisons, and mathematical operations with the input context elements.

(iii) *Medium:* represented by an $O(n^2)$ algorithm and implemented by a selection sort algorithm which sorts a list of $n$ real numbers. This load is intended to represent adaptation algorithms that do more sophisticated operations to the context elements.

(iv) *High:* represented by an $O(\log(n)n^2)$ algorithm and implemented by a selection sort algorithm done $\log(n)$ times. An algorithm with load of HIGH is very bulky and does high and complex number of operations, represents an extreme case.

In addition, to associate the algorithm load concept with the number of context elements which are the inputs of the adaptation algorithm (NUM_CONTEXTE), as well as with the number of adaption actions output to the algorithm (NUM_AACTIONS), we defined $n$ as indicated in (3), where $m = 100$ represents a base size of a list containing floating point numbers:

$$n = m + \frac{m}{10}\,(\text{NUM\_CONTEXTE} + \text{NUM\_AACTIONS})\,. \tag{3}$$

The execution times of these different adaptation configurations are shown in Figure 17; each measurement is the average of running it 100 times and includes the confidence interval of 95%. The charts (a) and (b) show the processing times measurements of the send, forward, and receive adaptation modes: in (a), by fixing the number of context elements to 5, then varying the load of the adaptation algorithm; in (b), by fixing the algorithm load to NONE, then varying the number of context elements used as input to the adaptation algorithm. The SF, SS, RF, RS, FF, and FS abbreviations in the horizontal axis of (a) and (b) refer to a combination of the adaptation mode and the special adaptation action used in the adaptation configuration; the first letter refers to the adaptation mode where the configuration is executed (send, receive, and forward), and the second letter refers to the special adaptation action we put in the configuration as output to the adaptation algorithm (filter and switcher).

The charts (c) and (d) of Figure 17 show the processing time measurements of the CUPIB adaptation mode. The chart (c) shows the processing times by varying the number of adaptation actions and fixing the algorithm load to NONE and the context elements to one. Chart (d) shows the processing times by varying the number of contexts elements of the algorithm, then fixing the algorithm load to NONE and the number of adaptation actions to one.

Examining the results of the processing cost measurements of the adaptation middleware, we concluded the following.

(i) The send and receive modes have very similar processing times; this is due to their similar implementations in the prototype. The forward mode processing times are a little higher than the send and receive modes, approximately 0.04 ms more.

(ii) The processing measurements in all the adaptation modes shown in the charts are located between 0.1 to 0.6 ms. The highest processing time obtained in the measurements was 1.6 ms of in CUPIB adaptation mode, with an algorithm load in HIGH, 9 context elements as input, and 9 adaptation actions as output; this measurement does not appear in the charts of Figure 17.

(iii) The algorithm load HIGH greatly elevates the processing times several tenths of a millisecond.

(iv) The processing time of all the adaptation modes when increasing the number of input context elements in the algorithm only makes increments of a few hundredths of milliseconds per context element.

(v) In the CUPIB mode, the processing time increments are more steep in relation with the number adaptation actions as output that when we add more context elements in the input of the adaptation algorithm.

Additionally, using the previous measurements framework we estimate the processing cost for the Case Study 1 example, which uses one context element as input, one adaptation action as output, and an algorithm load of LOW because the algorithm is a simple mapping of speed to

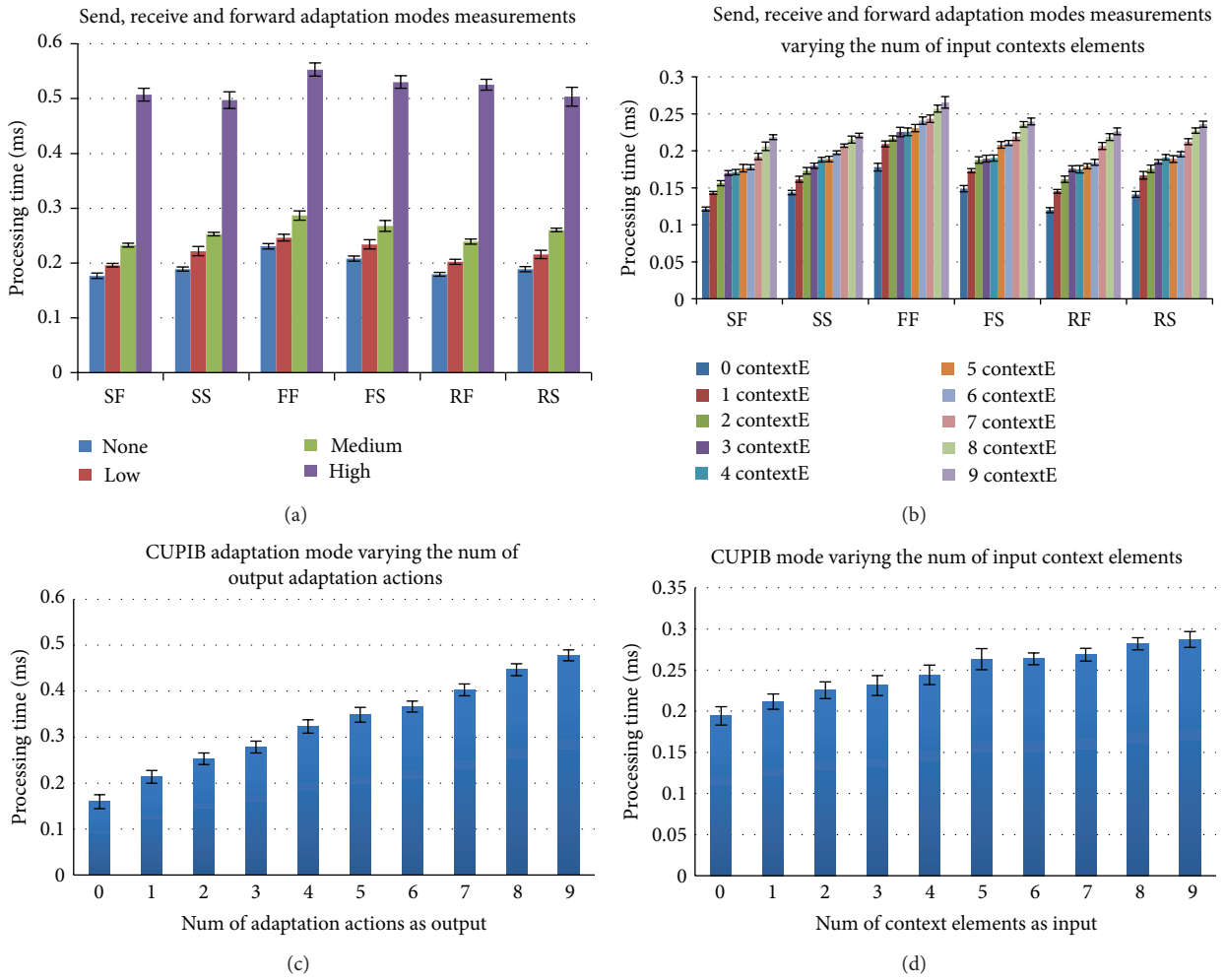


Figure 17: Measurement results of the processing overhead of the adaptation middleware prototype.

HELLO interval values. The processing time for Case Study 1 example resulted in 0.191 ms; this latency was introduced in the simulation of Case Study 1 in Section 6.2.

The measured time to initiate the middleware resulted in 654 ms. The initiation step covers from when the adaptation middleware bundle is started in Apace Felix to when the NL protocols, applications, and adaptation solutions in all the adaptation modes are started by the middleware.

Finally, based on the processing delay measurements of the adaptation modes (presented in Figure 17), we described the end-to-end delay incurred by the PLUGAM middleware. For each adaptation mode the end-to-end delay is affected in the following way.

(i) In case of executing an adaptation solution in the CUPIB mode, the end-to-end delay of a message is not affected because the execution of CUPIB is concurrent to the NL protocols, vehicular applications, and message passing between them.

(ii) In all the end-to-end path of a message, the send mode only adds a processing delay in the source node.

(iii) As for the receive mode, through all the end-to-end path of a message, this mode only adds a processing delay in the destination node.

(iv) In forward mode, the adaptation solution is executed in each node of the end-to-end path which relays the message; hence, the processing delay of the forward mode must be multiplied by the number of relay nodes of the path, in order to get the end-to-end delay of the middleware.

Figure 3 of Section 3.3 is useful to understand the end-to-end delay of the send, receive, and forward modes, because it shows their location in the end-to-end path of a message.

We present an analytical estimation of the end-to-end delay overhead of the middleware for two common types of NL protocols: unicast and dissemination. We assume that only one adaptation solution is defined either in the send, receive, or forward adaptation modes. These estimations are shown in Table 2. For the unicast estimation, we assume that the end-to-end route of the message consists of $R$ nodes. For the dissemination protocol estimation, we defined RN as the number of nodes participating in the retransmission of the

Table 2: Estimation of end-to-end delay of PLUGAM middleware for a general unicast and dissemination type NL protocol.

| Adaptation mode | Unicast protocol | Dissemination protocol |
|---|---|---|
| Send | SC $*$ ($R$) + CS | SC $*$ (DR + $D$ + 1) + CS |
| Receive | SC $*$ ($R$) + CR | SC $*$ (DR + $D$ + 1) + CR $*$ $D$ |
| Forward | (SC + CR) $*$ $R$ | SC $*$ (DR + $D$ + 1) + CF $*$ DR |

message (this number depends on the specific dissemination protocol) and DN as the number of destination nodes that transfer the message to the vehicular application.

In addition, SC refers to the processing cost of the search operation for an active adaptation configuration in the adaptation mode. This processing cost must be made even if no configurations are defined in the adaptation mode by the user. CS stands for the processing cost of executing an adaptation configuration defined in the send mode, CR in the receive mode, and CF in the forward mode.

*6.1.3. Plug-In Platform Memory and Processing Overhead.* A major aspect to consider about our proposed middleware architecture is the impact and feasibility of introducing a plug-in platform to a software system such as a middleware for vehicular systems, being that the plug-in platforms found in the software community are complex systems by themselves. We are particularly interested to see if adding a plug-in platform to communicate the internal middleware components does not add enough latency to the process.

In this particular plug-in platform implementation based on OSGi and iPOJO, in addition to the code written by the developer to implement a plug-in interface, in the compilation step extra code is automatically injected to the CLASS files in order to support the plug-in platform. This obviously increases the size of the plug-in binary (the JAR file) created by the developer. To measure this aspect, we took the ROM measurements from five plug-ins developed for the implementation of the Case Study 1 in the prototype; then we compiled them without OSGi and iPOJO functionality, only with OSGi functionality (making it an OSGi bundle), and with OSGi and iPOJO functionality. From these measurements, we found that the OSGi overhead code is small, in average only adding 345 Bytes to the plug-in binary; this small value is due to the fact that creating an OSGi bundle involves only adding some OSGi metainformation to the manifest file inside the JAR file.

In regard to the iPOJO code overhead, we found that it adds an average of 3.5 KB of code to the plug-in binary. Moreover, it increases the size of the class file implementing the iPOJO service by 86.84%; this code overhead originates because adding iPOJO functionality involves an injection of iPOJO specific methods to the original CLASS file; this code injection is done automatically by the Apache Ant compiler.

As we mentioned in previous sections, the plug-in platform communicates the extension components (added NL protocols, applications, etc.) to the middleware, this communication using the plug-in platform adds a processing overhead or extra latency that we think is interesting to measure. To calculate this latency, we recreated a scenario in which we have two iPOJO components A and B, where A uses the B component and both components are located in different bundles or JAR files. Then at run time, we measured the time between when component A calls the method of component B and until it immediately enters the method of component B. The resulted latency was 0.009 ms, which is a very small latency. However, we noticed a bigger latency (1.74 ms) when an iPOJO method is called for the first time; this behavior occurs because at run time iPOJO delays loading the iPOJO components until the component methods are actually called by another and not at initiation time (iPOJO calls this "lazy object creation").

Finally, we measured the initiation time of the prototype's plug-in platform; we measured the time since the Apache Felix OSGi container is started by calling the "java -jar felix.jar" command until when Apache Felix loads the PLUGAM plug-ins and gives execution control to the adaptation middleware. The measured time was 2.4 seconds, from which 0.986 ms are needed by Apache Felix to start the OSGi container, and the rest is spent on loading the plug-ins from a specific folder of Apache Felix.

In summary, the code required to encapsulate the extension component functionality into a plug-in is small, a few KB, and most of this extra code comes from iPOJO. In contrast, the extra code required to create an OSGi bundle is minimum. Moreover, this plug-in platform implementation is very light and adds very little latency when communicating the plug-ins with the middleware; however, the first time these two modules communicate generates a significant latency. Furthermore, the 2.4 s initiation time of this plug-in platform implementation is lengthy, which in addition to the middleware's initiation time (presented in Section 6.1.2) sums up a total of approximately 3 seconds of initiation time before the NL protocols and applications start working. This start-up delay must be taken into account when deploying the vehicular system in the target vehicles.

*6.2. Evaluation of Case Study 1.* In this section, we present an evaluation of the adaptive protocol proposed for the Case Study 1 example throughout simulation. The idea of this evaluation is to show a brief quantitative evaluation on the impact of specific proposed adaptation, which is built with our middleware. We want to assess if even this simplistic adaptation proposal indeed generates a performance increase as reported by Huang et al. [26].

Moving on to the details of this evaluation, essentially we compared two protocols, the default OLSR protocol and the adaptive OLSR protocol which includes the adaptation proposed for Case Study 1. Both are based on UM-OLSRv0.8.8 (http://masimum.inf.um.es/fjrm/?page_id=116), an OLSR implementation for NS-2. We measured the following metrics in the simulation to compare the performance of the two protocols.

(i) *Average goodput* of all data connections in the simulation: the goodput is defined as the amount of

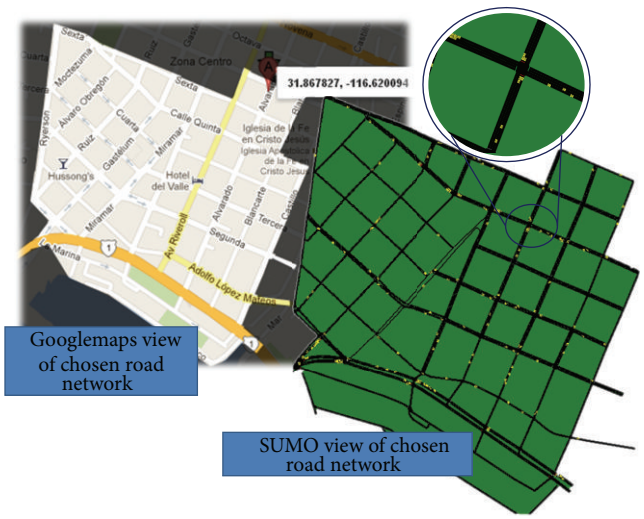


Figure 18: Chosen road network for the simulation of Case Study 1.

useful data delivered (in bytes) divided by the data connection transfer time.

(ii) *Normalized routing overhead* (NRO), which is defined as the ratio between the number of control packets propagated by every node in the network and the number of data packets received by the destination nodes. This metric is strongly related to the control packet overhead of the NL protocol.

We also measured the average end-to-end delay; however, the resulted delay values were too scattered, thus not useful when comparing the two protocols; this can be explained because the OLSR protocol itself is not sensitive to delay.

For the NS-2 simulation, the speed context element for each node is obtained from the speed value in the MobileNode class in the NS-2 code; this speed value has 100% precision contrary to the speed values obtained from a real world GPS. Moreover, instead of porting the entire middleware to NS-2, we only added the CUPIB mode functionality to the adaptive OLSR protocol; more specifically, we added a delay of 0.1916 ms obtained from the cost evaluation of the prototype, in order to emulate the processing overhead of the adaptation middleware when executing the CUPIB mode.

In addition, to simulate a city scenario we used real data (roads specification) concerning an urban area of the city of Ensenada in Mexico, this data was exported from openstreetmap (Openstreetmap: http://www.openstreetmap.org/) and it is shown in Figure 18. For the simulation of vehicles and their movements (considering traffic lights), we used the SUMO tool (Simulation for Urban Mobility: http://sumo.sourceforge.net/). We generated the flow of vehicles from 41 routes in the map by defining the begin and end points of the routes. The internal vertices of routes were generated automatically by the SUMO Duarouter program that uses Dijkstra's shortest route algorithm, where the edge weights are a combination of the max speed of a street and the distance to the destination and also from traffic situation. To emulate real world vehicle movements, the vehicles can enter and get out of the simulation area. The city max speed and hence the vehicles max speed was set to 40 kph. The wireless range were set to 150 m, instead of the default 250 m of 802.11, in order to emulate the buildings of the city center.

Table 3: Simulation of Case Study 1 parameters.

| Simulation parameters | |
|---|---|
| Simulator | NS-2.34 |
| Simulation time | 100 seconds |
| Simulation area | Approx 1 km × 1 km |
| MAC layer | IEEE 802.11b |
| Traffic type | CBR/UDP |
| CBR data payload | 512 KB |
| CBR rate | 100 kps |
| Number of runs | 5 |

Furthermore, we compared the two protocols in different scenarios varying the global network density (GND) and average global node speed (GNS). By global network density we mean the total number of vehicles in the entire simulation time; the network density was varied between 100, 150, and 250 vehicles. We only simulated up to 250 vehicles due to computational resources and time constrains. For example, doing a simulation run with 250 vehicles took around 5 days and simulating 350 vehicles generated simulation runs taking a whole month. Extending the simulations results with higher GND values is desirable thus regarded as future work.

As for the average node speed, we are referring to the average speed of all the vehicles in the network, during all the simulation time; we present results with GNS of 10, 15, 20, 25, and 30 kph. The reason the speed values were so small is because the vehicles spent a lot of time in stops and traffic lights. In addition, we generated the maximum feasible number of CBR connections between the vehicles in the simulation area, which is less than half the total number of nodes in the simulation, and the CBR connections were active during all the simulation time. Additional simulation parameters are shown in Table 3.

In Figure 19, we present the results comparing the adaptive OLSR versus the normal OLSR. In the upper charts, we measured the goodput of the two protocols with respect to the various speed (GNS) values, and we did this for each of the three node density (GND) scenarios taken into account. In the lower charts, we measured the normalized routing overhead (NRO) between the two protocols with respect to the different values of GNS.

Concerning the goodput metric, we can say that the goodput of the adaptive OLSR closely mirrors that of the normal OLSR, nonetheless generating less goodput. Doing an overview analysis of all the measurements with the different GNS and GND values, we concluded that in average the adaptive OLSR goodput is 1.42% less than the normal OLSR; this means that the adaptation of Case Study 1 generated worse performance than the normal OLSR. However this conclusion lacks precision because the resulting confidence intervals are not small enough. Increasing the number of runs is needed to provide this certainty.

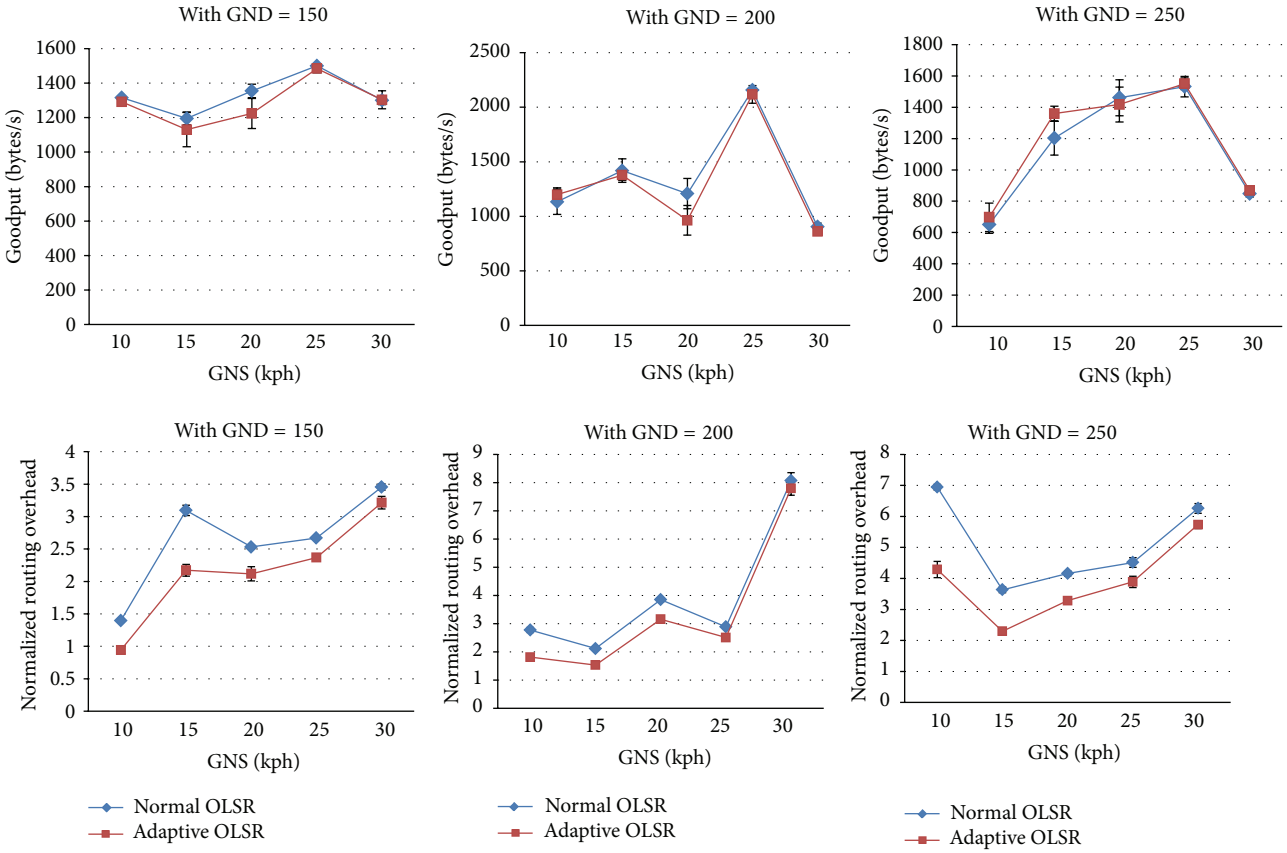


Figure 19: Simulation results of Case Study 1.

Regarding the NRO metric, we can see a clear advantage of the adaptive OLSR in every scenario because it produces less control packets. In this case the confidence intervals are small enough to provide us with this certainty in spite of only doing five runs. Nevertheless, we notice that the bigger the speed (GNS) value, the advantage of the adaptive OLSR begins to diminish. Doing an overview analysis of all the measurements with the different GNS and GND values, the adaptive OLSR NRO value was an average of 20.78% less than normal OLSR; this means that the adaptation of Case Study 1 generates 20% less control packages given that both generated almost the same goodput.

In summary, the adaptation of Case Study 1 generated an improvement by reducing by 20% the number of control packages of OLSR, at the cost of reducing its goodput by approximately 2%.

*6.3. Scalability of the PLUGAM Architecture and Comparison against an Existing Adaptive Protocol.* Finally, to get a better grasp of the performance and efficiency of the adaptation middleware architecture, in this section we discuss its scalability and make a comparison with an existing adaptive protocol.

We refer to the scalability of the PLUGAM in two aspects: firstly based on the number of nodes in the network, and secondly based on the number of NL protocols used in the middleware instance by the user. Regarding the scalability with respect to the number of nodes/vehicles in the network, we observed the following.

(i) Each node in the network has one middleware instance installed, each one contains the same context elements, NL protocols, plug-ins, adaptation solutions, applications, and so forth. The current focus of this architecture is offering only node-local adaptations, this means that PLUGAM architecture is a purely distributed system, which in principle has better scalability than a centralized system. The definitions of node-local and distributed adaptations can be found in Section 2.

(ii) If the architecture was to be extended to deal with distributed adaptations (regarded by us as future work), then it would not be a purely distributed system anymore. Distributed or global adaptations need centralization or additional interactions between nodes to coordinate adaptations, this would surely impact the scalability with respect to the number of nodes.

Regarding the scalability with respect to the number of NL protocols used in PLUGAM middleware, the generic aspect of the architecture generates extra overhead in contrast to particular adaptive protocol solutions, every NL protocol added generates the following extra costs.


Figure 20: Continued.

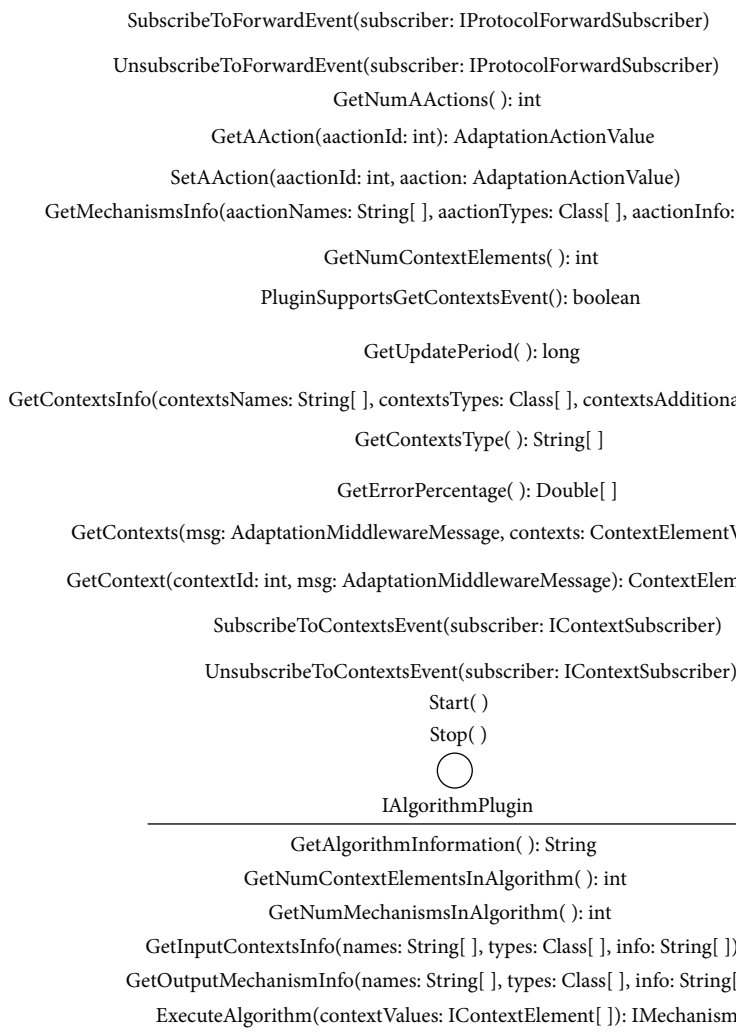


Figure 20: Definition of the service interfaces for all the different types of plug-ins in the PLUGAM architecture.

(i) Each NL protocol, same as the other extension components of the architecture (context elements, vehicular applications, adaptation configurations, etc.), incurs additional memory overhead due to the need to attach metainformation that the middleware uses to manage them. This is the cost of modularization and low coupling design of the extension components.

(ii) Each NL protocol, same as the other extension components of the architecture, is encapsulated in a plug-in. To create a plug-in extra special code and instructions need to be added in order to enable a communication channel with the plug-in platform. In addition, extra memory overhead is needed in order to implement the protocol plug-in interface. Measurements for Section 6.1.3 indicate around a 3.5 KB increase in code size.

Related to having a high number of adaptation solutions in the middleware, this increases the probability of having adaptations interfere with each other producing unexpected results; thus, special care and experimentation are needed to calibrate the effects of the adaptation solutions added in them. The scalability of the architecture is also dependent on the plug-in platform efficiency in regard to loading a high number of plug-ins, especially in the initiation step in which all the plug-ins are loaded (our prototype gave us initiation times of around two seconds). In the case of the prototype's plug-in platform based on OSGi/iPOJO, no serious scalability analysis has been found in the literature; however, OSGi has been used to build programs (e.g., Eclipse) supporting a high number of modules.

Furthermore, to get a more comprehensive view of the performance and efficiency of PLUGAM we emphasize on the extra overhead generated by our generic solution in contrast with a particular adaptive protocol. We will use as example the proposal of Chen et al. [14] which performs an adaptation based on switching between two different NL protocols (more information can be seen in Table 4 of Section 7). The extra overhead generated by our generic solution is the following.

Table 4: Comparative analysis of some adaptive protocols found in the literature.

| Adaptive protocols | Field of research | Context elements used | Adaptation actions used | Adaptation algorithm used | Adaptation modes used |
|---|---|---|---|---|---|
| Adler et al. [3] | VANETs | No specific context elements, general approach by relevance parameters | Forward decision, priority to send message, and special filter adaptation action | Message relevance function defined by authors | Send, receive, and forward |
| Delot et al. [6] | VANETs | Mobility and direction vector of event and vehicle | Forward decision, use of filter special adaptation action | Encounter probability function defined by authors | Forward |
| Ramasubramanian et al. [7] | MANETs | Perceived packet overhead, packet loss rate, and jitter between neighbor | Zone radius of each destination | Three heuristics to calculate each perceive metric, then modify the zone radius when certain threshold is reach | CUPIB |
| Giannoulis et al. [8] | Ad hoc networks | Routing failure and number of nodes in zone | Zone radius and route update interval | Decision to modify radius zone if result is outside of threshold | CUPIB and send |
| Zhao and Cao [9] | Ad hoc networks | No specific context elements, only gives examples like degree to which channel is busy | Selecting routing module, tuning routing algorithm parameters, and adjusting routing metric | Doesn't have specific algorithms, gives examples of threshold algorithms | CUPIB |
| Abuelela and Olariu [10] | VANETs | Local traffic conditions | Switch between mulling or unicast routing protocols | By detecting presence of path to next oncoming cluster on the road | Forward and send |
| Eichler et al. [11] | VANETs | No specific context elements, general approach by context parameters | Priority to send message | Message benefit function defined by authors | Send |
| Al-Doori et al. [12] | VANETs | Road direction and vehicles velocity | Broadcast HELLO message | Simple threshold between velocity values and detection algorithm of change of direction | CUPIB |
| Mariyasagayam et al. [13] | VANETs | Neighbor density | Forward decision, use of the filter special adaptation action | Forwarding sectors algorithm | Forward |
| Chen et al. [14] | VANETs | Time of traffic lights, max speed of road, speed of neighbor vehicles, and the predicting slope | Forward decision, use of the switcher special adaptation action | Algorithm based on value of predicting slope and three scenarios of traffic light | Forward |

(i) An additional processing cost exists due to the indirect access between the extension components involved in the adaptation solution, contrary to particular adaptive protocols like Chen et al. where these elements are tightly integrated and directly communicating. This processing cost mostly depends on the communication delay between plug-ins and the middleware passing through the plug-in platform, in our prototype implementation this delay is very small, around 0.009 ms as seen in Section 6.1.3.

(ii) The overhead generated by the middleware header (132 bytes) must be taken into account because it lowers the throughput of the protocols by taking bytes for useful data. The middleware header helps to support a multiple-applications multiple NL protocols environment and identifies the NL protocol and application attached to the message. The work of Chen et al. [14], not being a generic approach, treats these two NL protocols as a single super protocol and hence does not need this header.

Table 5: Definition of the ElementIdentifier class (a) and the meaning of its attributes (b).

(a)

| ElementIdentifier |
|---|
| (i) name: String |
| (ii) hash: long |
| (iii) secondName: String |

(b)

| ElementIdentifier instance | |
|---|---|
| name | NP |
| hash | HP |
| SecondName | NE |

Table 6: Contents of an adaptation configuration.

| Adaptation configuration | |
|---|---|
| Field | Data type |
| configurationName | String |
| adaptationAlgorithm | ElementIdentifier |
| contextElements | List of ElementIdentifier |
| adaptationActions | List of ElementIdentifier |
| adaptationMode | {"Send", "Receive", "Forward", "CUPIB"} |
| protocol | ElementIdentifier |
| application | ElementIdentifier |
| CUPIBTime | Long integer |

(iii) PLUGAM architecture generates a processing delay to setup and execute an adaptation solution; a particular solution like Chen et al. has direct access to the context elements values and only executes the adaptation algorithm. From Section 6.1.2, the processing time to setup and execute an adaptation solution in the prototype is inferred to be less than 0.2 ms.

(iv) Particular adaptation solutions like Chen et al. do not include a concurrent processing overhead generated by the context manager subsystem.

## 7. Related Work

There have been several adaptive protocols proposals for mobile networks which have shown their efficiency in dealing with changing environments. In Table 4, we present a few of these adaptive protocols by highlighting their association to their possible implementation using our adaptation middleware architecture.

We found that most of these adaptive protocol proposals were self-contained, meaning that the proposals are usually promoted as another novel network layer protocol which handles better the dynamism of the network, nevertheless no work is done to relate their adaptation solutions to other proposals nor to propose a common adaptation model; this makes it difficult to promote the reuse of parts of their solution or ideas. Furthermore, we also notice that their adaptation solutions are very specific to a particular NL protocol. That is why the aim of our work is to promote the generalization of adaptation solutions independent of the NL protocols and context variables by means of proposing a subsystem in the form of a middleware and based on an adaptation model.

Besides the adaptive protocols found in MANETs and VANETs, there is some related work on subsystems or middleware proposals aiming to improve the performance of NL protocols and applications to dynamic conditions.

From the point of view of using context information to produce more dynamic applications and communications, Peddemors et al. [28] proposed a platform to express and provide context information to mobile applications; the platform allows applications to read the context variables or act upon them by adapting their behavior. Salber et al. [24] proposed the use of a context widget to represent and provide applications access to context information. The context widget platform provides a standard subscription mechanism and a polling mechanism. The context widget architecture includes composition of context information or context fusion acquired from multiple distributed sources.

The previous context platforms only focused on offering context to applications and not to network layer protocols as we are proposing, and the authors gave no further details on how to perform the adaptation of the application behavior once the context information is obtained. However, their context model is more complex than ours.

On the other hand, there are other subsystem type proposals which offer a way to adapt the communication system by switching network protocols or reconfiguring them from within (using a compositional adaptation approach). For example, Nundloll et al. [5] proposed a framework for run time reconfiguration of routing protocols for VANETs. This framework encapsulates a network layer protocol in a component and also separates it in subcomponents using a control-forward-state (CFS) pattern. This framework adapts or reconfigures the protocols by making changes at two levels: add/remove the protocol components (switching NL protocols) or change CFS subcomponents within the protocol. The authors also identified some common network layer protocol operations where adaptations can be performed; these operations are the send, receive messages, scheduling tasks, and neighbor discovery. This framework for protocol reconfiguration has also been applied to other types of networks (wireless sensor networks [29] and MANETs [30]). Furthermore, MANET researchers have developed a number of reconfigurable ad hoc protocol frameworks, prominent among which are ASL [31] and PICA [32].

Comparing these compositional adaptation proposals to our adaptation middleware, these works require a full reimplementation of NL protocols to take advantage of interchanging their internal components at run time. In contrast, our adaptation middleware architecture is based on the parameter adaptation approach, and porting already existing NL protocols is more simple because it only requires to

implement the protocol plug-in interface. Another difference is that our middleware proposes the use of multiple NL protocols running concurrently in the network stack, contrary to these other works which only consider having one active NL protocol at a time. Finally, Nundloll et al. [5] framework proposal also does not offer a concrete solution on how to express and use the obtained context information, as does our middleware solution.

## 8. Conclusions and Future Work

We have presented PLUGAM, a middleware that handles the adaptation of the network layer protocols in a vehicular system; the aim of this paper was to explain the middleware architecture in detail. In particular, a novel modeling of the adaptation concept applied to NL protocols is presented, and this model is used as the basis of the middleware. The adaptation middleware is a general solution to help build a set of adaptive protocols found in the literature. Not all the adaptive protocol proposals can be built with our architecture; however, the architecture is flexible enough to be easily extended by adding more adaptation modes, and special adaptation actions. The definition of the architecture is guided by the provision of two key types of adaptations to NL protocols: one is to modify the NL protocol behavior by adaptation based on context information of the node, and the second is to propose a multiple NL protocols environment (where more than one are running concurrently in the system) and allowing adaptations where a selection between them can be made based on context information.

Furthermore, we assess the feasibility of our approach with two distinct case study examples of adaptive protocols which can be built with the middleware, and each represents the key types of adaptations mentioned previously. We also conducted via simulation a short analysis of the performance of the Case Study 1 adaptation in order to assess the reported improvements. Although the proposed Case Study 1 adaptation was far too simplistic, the results show that the adaptive version of OLSR produces 20% less control overhead than normal OLSR, and only lowering the goodput by 2% to that of Normal OLSR. In addition, to demonstrate feasibility of the middleware architecture, we implemented a prototype along with a set of plug-ins examples, our main concern was to see if using a plug-in platform to interconnect the extension components of an adaptive protocol is a feasible solution. The chosen plug-in platform for the prototype was implemented using OSGi and iPOJO technologies.

To get a rough idea on the cost and performance of the adaptation middleware, we presented some measurements done to the prototype and analyzed the memory and processing overhead of using the middleware. Results show that the prototype's RAM (20 MB) and ROM (111 MB) usage is small enough to make it viable to implement the adaptation middleware in hardware similar to today's medium entry level smartphones having a Java virtual machine. In terms of processing overhead of the prototype, the results indicated that the latency to execute an adaptation solution in each of the adaptation modes is around 0.5 to 1.5 ms, which in terms of network protocols is acceptable. These times were measured on a 1.6 GHz dual core processor, which corresponds to the processing power of today's high-end smartphones; this means that to guarantee similar latencies the future vehicular systems need at least this processing power. We think that vehicular systems with similar hardware specs are feasible to appear in a few years.

We also analyzed the cost/performance of only the plug-in platform of the prototype. The analysis shown, in particular that of OSGi and iPOJO technologies, has a very small processing overhead; we think this was the main reason of the small processing overhead results of the adaptation middleware prototype. Although we have noticed that other plug-in platforms are not as lightweight as the OSGi and iPOJO technologies.

The adaptation middleware architecture we have presented is part of an ongoing project, and we are working to accommodate a wider range of adaptive protocols and improve the basic context and multiple NL protocols environments. Specific aspects we want to explore and thus regarded as future work are the following.

(i) The adaptation middleware is limited to building node-local adaptations, as future work we want to explore extending the architecture to also allow distributed or global scope adaptations.

(ii) Explore using a more complex context environment, in particular explore the use of context ontologies to allow the declaration of a type of context element to fit an input slot of an adaptation algorithm without specifying the context source; also explore the fusion of context information to produce new context information, and finally add context information history to refer to the last values of a context element.

(iii) Introduce message-specific context elements in which the current value of the context is tied to a message. For example, have a priority of the message context element and have the location context information of the node that relays the message.

(iv) Introduce mechanisms to validate the run time effects of an adaptation in the system, or at least provide some feedback of the effects. This could help speed up the calibration of adaptation solutions built with our middleware in order to improve performance.

(v) A mechanism to deal with possible NL protocols interference if they use the same communication channel.

(vi) Propose a general node addressing scheme to unify or translate source and destination addresses between the different types of NL protocols supported by the middleware (to remember why this is useful see Section 3.7).

(vii) Introduce new adaptation modes; for example, one that executes an adaptation solution in an NL protocol or vehicular application event, or when certain values of context elements are met.

## Appendix

## Plug-In Type Interface Definitions

In this appendix we present the definition of all the different plug-in interfaces in the architecture which were mentioned at the end of Section 3.4. These interface definitions are shown in Figure 20. The IContextEPlugin interface contains the methods to build a contextE plug-in. The IProtocolPlugin interface contains the methods to build a protocol plug-in. The IAlgorithmPlugin interface contains the methods to build an algorithm plug-in. And finally, the IApplicationPlugin interface contains the methods to build an application plug-in.

## Conflict of Interests

All the funding sources for this work are listed in the acknowledgement section of the paper. None of the authors of this work have a significant financial relation or affiliation with any product or trademark mentioned, nor any potential bias against another product.

## Acknowledgments

Partial funding for this work was provided by the Mexican Council for Science and Technology (CONACyT). This work is also funded by the ANR research agency (France) within the scope of the Optimacs project.

## References

[1] S.-h. An, B.-H. Lee, and D.-R. Shin, "A survey of intelligent transportation systems," in *Proceedings of the 3rd International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN '11)*, pp. 332–337, 2011.

[2] E. Schoch, F. Kargl, M. Weber, and T. Leinmüller, "Communication patterns in vanets," *IEEE Communications Magazine*, vol. 46, no. 11, pp. 119–125, 2008.

[3] C. Adler, S. Eichler, T. Kosch, C. Schroth, and M. Strassberger, "Self-organized and context-adaptive information diffusion in vehicular ad hoc networks," in *Proceedings of the 3rd International Symposium on Wireless Communication Systems (ISWCS '06)*, pp. 307–311, September 2006.

[4] A. Festag, G. Noecker, M. Strassberger, A. Lübke, and B. Bochow, "Now—network on wheels: project objectives, technology and achievements," in *Proceedings of the 5th International Workshop on Intelligent Transportation (WIT '08)*, pp. 211–216, March 2008.

[5] V. Nundloll, G. S. Blair, and P. Grace, "A component-based approach for (re)-configurable routing in vanets," in *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware (ARM '09)*, vol. 2, ACM, New York, NY, USA, 2009.

[6] T. Delot, N. Cenerario, and S. Ilarri, "Vehicular event sharing with a mobile peer-to-peer architecture," *Transportation Research Part C*, vol. 18, no. 4, pp. 584–598, 2010.

[7] V. Ramasubramanian, Z. J. Haas, and E. G. Sirer, "SHARP: a hybrid adaptive routing protocol for mobile ad hoc networks," in *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing (MobiHoc '03)*, pp. 303–314, ACM, New York, NY, USA, June 2003.

[8] S. Giannoulis, C. Katsanos, S. Koubias, and G. Papadopoulos, "A hybrid adaptive Routing protocol for Ad hoc wireless networks," in *Proceedings of IEEE International Workshop on Factory Communication Systems (WFCS '04)*, pp. 287–290, September 2004.

[9] J. Zhao and G. Cao, "VADD: vehicle-assisted data delivery in vehicular ad hoc networks," in *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM '06)*, April 2006.

[10] M. Abuelela and S. Olariu, "Traffic-adaptive packet relaying in VANET," in *Proceedings of the 4th ACM International Workshop on Vehicular Ad Hoc Networks (VANET '07)*, pp. 77–78, New York, NY, USA, September 2007.

[11] S. Eichler, C. Schroth, T. Kosch, and M. Strassberger, "Strategies for context-adaptive message dissemination in vehicular ad hoc networks," in *Proceedings of the 2nd International Workshop on Vehicleto Vehicle Communications*, pp. 1–9, July 2006.

[12] M. M. Al-Doori, A. H. Al-Bayatti, and H. Zedan, "Context aware architecture for sending adaptive HELLO messages in VANET," in *Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems (CASEMANS '10)*, pp. 65–68, ACM, New York, NY, USA, September 2010.

[13] N. Mariyasagayam, H. Menouar, and M. Lenardi, "An adaptive forwarding mechanism for data dissemination in vehicular networks," in *Proceedings of the IEEE Vehicular Networking Conference (VNC '09)*, October 2009.

[14] Y. S. Chen, C. S. Hsu, and Y. T. Jiang, "A delay-bounded routing protocol for vehicular ad hoc networks with traffic lights," in *Proceedings of the IEEE International Conference on Communication, Networks and Satellite (ComNetSat '12)*, pp. 122–126, 2012.

[15] R. Rejaie, M. Handley, and D. Estrin, "Quality adaptation for congestion controlled video playback over the internet," *SIGCOMM*, vol. 29, no. 4, pp. 189–200, 1999.

[16] Q. G. K. Safi, T. Nawaz, S. M. A. Shah, and T. Mahmood, "Intelligent device independent ui adaption for heterogeneous ubiquitous environments," *IJCSNS International Journal of Computer Science and Network Security*, vol. 11, no. 11, 2011.

[17] R. Ramdhany, P. Grace, G. Coulson, and D. Hutchison, "Manetkit: supporting the dynamic deployment and reconfiguration of ad-hoc routing protocols," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware '09)*, Springer, New York, NY, USA, 2009.

[18] D. Popovici, M. Desertot, S. Lecomte, and N. Peon, "Context-aware transportation services (cats) framework for mobile environments," *International Journal of Next-Generation Computing*, vol. 2, no. 1, 2011.

[19] P. Grace, "Dynamic adaptationin," in *Middleware for Network Eccentric and Mobile Applications*, H. M. B. Garbinato and L. Rodrigues, Eds., pp. 285–304, Springer, 2009.

[20] B. D. Win, F. Piessens, W. Joosen, and T. Verhanneman, "On the importance of the separation-of-concerns principle in secure software engineering," in *ACSA Workshop on the Application of Engineering Principles To System Security Design*, C. Serban, Ed., pp. 1–10, 2003.

[21] J. Boleng, W. Navidi, and T. Camp, "Metrics to enable adaptive protocols for mobile ad hoc networks," in *Proceedings of the International Conference on Wireless Networks (ICWN '02)*, pp. 293–298, June 2002.

[22] C. Yawut, B. Paillassa, and R. Dhaou, "Mobility metrics evaluation for self-adaptive protocols," *Journal of Networks*, vol. 3, no. 1, pp. 53–64, 2008.

[23] I. Hadzic, W. S. Marcus, and J. M. Smith, "Policy and mechanism in adaptive protocols," Tech. Rep. MS-CIS-01-03, Department of Computer and Information Science University of Pennsylvania and Bellcore, 1999.

[24] D. Salber, A. K. Dey, and G. D. Abowd, "Context toolkit: Aiding the development of context-enabled applications," in *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit (CHI '99)*, pp. 434–441, ACM, New York, NY, USA, May 1999.

[25] R. Wolfinger, D. Dhungana, H. Prähofer, and H. M. Mössenböck, "A component plug-in architecture for the .net platform," in *Modular Programming Languages, Ser. Lecture Notes in Computer Science*, D. Lightfoot and C. Szyperski, Eds., vol. 4228, pp. 287–305, Springer, Heidelberg, Germany, 2006.

[26] Y. Huang, S. N. Bhatti, and D. Parker, "Tuning olsr," in *Proceedings of the IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 06)*, pp. 1–5, September 2006.

[27] G. L. Pedro, G. Raúl, E. Marc, P. Gerard, A. Marcel, and M. Roc, "Topology-aware group communication middleware for MANETs," in *Proceedings of the 4th International ICST Conference on COmmunication System softWAre and middleware (COMSWARE '09)*, vol. 7, pp. 978–971, ACM, Dublin, Ireland, 2009.

[28] A. Peddemors, H. Eertink, and I. Niemegeers, "Communication context for adaptive mobile applications," in *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom '05)*, pp. 173–177, March 2005.

[29] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes, "Dynamic reconfiguration in sensor middleware," in *Proceedings of the international workshop on Middleware for sensor networks (MidSens '06)*, pp. 1–6, ACM, New York, NY, USA, 2006.

[30] R. Ramdhany, P. Grace, G. Coulson, and D. Hutchison, "Manetkit: supporting the dynamic deployment and reconfiguration of ad-hoc routing protocols," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware '09)*, vol. 1, Springer, New York, NY, USA, 2009.

[31] V. Kawadia, Y. Zhang, and B. Gupta, "System services for ad-hoc routing: architecture, implementation and experiences," in *Proceedings of the 1st international conference on Mobile systems, applications and services (MobiSys '03)*, pp. 99–112, ACM, New York, NY, USA, 2003.

[32] C. M. T. Calafate and P. Manzoni, "A multi-platform programming interface for protocol development," in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '08)*, vol. 0, p. 243, 2003.