About efficiency in wireless communication frameworks on vehicular networks

Invited paper

Bertrand Ducourthial
Lab. Heudiasyc UMR CNRS 6599
Université de Technologie de Compiègne
Compiègne, France
bertrand.ducourthial@utc.fr

ABSTRACT

Nowadays, the Intelligent Transport Systems (ITS) attract many attentions. ITS applications would indeed increase the road safety and the transport efficiency, limit the impact of the vehicles on the environment, improve the overall productivity... However, several important open issues have to be solved before. Among them, the software architecture represents a key issue. Indeed, most of the ITS applications will rely on a distributed frameworks embedded in the vehicles. These applications often require robustness, quality of services or real time management, while the vehicular networks present important constraints in terms of dynamic, variable density, communication reliability, communication duration...

In this paper, we discuss the requirements of the software architectures for ITS applications. Next we propose an efficient architecture for distributed applications over vehicular networks, called Airplug. This architecture has been implemented in a light embedded framework, and has been used to test distributed services and protocols on the road. We then show that such a light and efficient architecture is well adapted to build distributed applications on vehicular networks.

Keywords

Vehicular networks, VANET, dynamic ad hoc networks, ITS, embedded architecture, distributed framework, road experiments

1. INTRODUCTION

1.1 Intelligent Transportation System

The Intelligent Transportation Systems are intended to improve the transportation in terms of safety, mobility, impact on the environment, productivity... The underlying technologies encompass a broad range of communications and electronics technologies. They are integrated in the transportation system's infrastructure as well as in vehicles themselves.

Among the applications, we may quote i) the *infrastructure oriented applications* for optimizing their management (transit management, freeway management, intermodal freight, emergency organization...) , ii) the *vehicle oriented applications* for increasing the road safety (incident management, crash prevention, collision avoidance, driver assistance...), iii) the *driver oriented services* for improving the road usage (traffic jam and road work information, traveler payment, ride duration estimate...) and iv) the *passengers oriented applications* for offering new services on board (Internet access, distributed games, chats, tourist information, city leisure information, movies announces downloads [21]...).

The ITS motivations are multiple. With a better resource management (infrastructure, car fleets, intermodal freight...), the transport productivity will increase. Regarding the road safety, the Department of Transport (DoT) of the USA launched large initiative to reduce the number of deaths in the road (around 43,000 per year) [8]. The European Commission (EC) targets to halve the number of road fatalities by 2010 [5] and it launched large ITS projects. Some of the ITS applications are studied by car manufacturers to propose more and more equipped vehicles. A new business related to on board services may appear in few years. Finally, a better road management either by the infrastructure or by the drivers will contribute to environmental preservation by avoiding traffic congestion, optimizing the car speed, easing public transportation (intermodality) or organizing car sharing services for instance. Since consumers are more and more concerned by safety and environmental issues, all these services became marketing arguments for car manufacturers.

1.2 Projects

Nowadays, ITS attract many attention of governmental agencies, industries and research teams all over the world. Large R&D initiatives have been launched in the USA (VII, CICAS, IVBSS...), in Europe (CVIS, SAFESPOT, COOPERS, PReVENT, GST, HIGHWAY, FLEETNET...), in Japan (SmartWay, VICS), in India (ITSIndia), in Germany (NoW), in France (PREDIT)... We highlight some of the main projects that require vehicle-to-vehicle (V2V) or vehicle-

to-infrastructure (V2I) communications.

In the USA, the DoT launched an important ITS program [8]. A functional architecture has been defined for the future national ITS applications [9]. The Cooperative Intersection Collision Avoidance Systems (CICAS) initiative aims at improving the road safety by enhancing driver decision-making at intersections [1]. It relies on V2I communications based on DSRC. The Vehicle Infrastructure Integration (VII) initiative focuses on the deployment of advanced V2V and V2I communications to increase road safety and relieve traffic congestion [13].

In Europe, major R&D projects are supported by the EC to constitute the basis of an European-wide intelligent transportation system. The COMeSafety project supports the eSafety Forum which is dedicated to the improvement of road safety using ITS. The Cooperative Vehicle Infrastructure Systems project (CVIS) focuses on the technologies that will enable V2V and V2I communications between the vehicles and the infrastructure [4]. The goal is to develop a cooperative road transport system, in order to increase the road safety and efficiency, and to reduce the environmental impact of the road transport on the environment [3]. The SAFESPOT project aims at developing a *Safety Margin Assistant* that will detect dangerous situations in advance, in order to extend the driver awareness of the surrounding environment [12]. Such assistant is based on V2V and V2I communications. The COOPERS project (Cooperative Systems for Intelligent Road Safety) focuses on the development of innovative telematics applications on the road infrastructure to enable cooperative traffic management on motorway section [2]. The goal is to enhance the road safety by providing direct and up to date traffic information to the motorized vehicles. The PReVENT project addresses the integration of new in-vehicles systems which sense the environment [11]. The goal is to help the driver to avoid or mitigate an accident. For this purpose, the WILLWARN subproject works on a decentralized warning distribution relying on V2V communication. The GST project (Global System for Telematics) focuses on the creation of an open and standardized end-to-end architecture for automotive telematics services [7]. With this open platform, vehicle manufacturers, public services and certified companies will allow to provide and distribute their own infrastructure-oriented services to consumers (eg. emergency call services, enhanced floating car data services, safety warning and information services...).

1.3 Embedded architectures

Almost all ITS applications will rely on software embedded in the vehicles, and many of them will require V2V or V2I communications, as the above described projects. To ensure the good working order of the applications and to launch the new ITS business, a standard embedded architecture is certainly welcome.

Standardization of the vehicular communication is now ongoing in major international organizations (IEEE, IETF, ETSI, ISO, SAE, ASTM), industrial consortia such as the Open Mobile Alliance (OMA) [10] and the Car-to-Car Communication Consortium (C2C-CC) [16] and national ITS authorities. The IEEE develops the Wireless Access in Vehicular Environments (WAVE) and extensions of the 802.11 protocols for ITS applications (IEEE 802.11p). The Continuous Air-Interface for Long and Medium range telecommunication standard (CALM) is developed by the ISO Technical Committee 204, Work Group 16 (ISO TC204 WG16). This standard assumes that future vehicles will be equipped with more than one wireless technology. CALM includes IPv6, and a set of protocols to route data over the available wireless technologies to ensure quality of service. The IETF develops extensions of IP for mobiles nodes (Mobile IP) and mobile networks (NEMO). These protocols are considered in CALM. The OMA develops protocols for data management among mobile nodes. The C2C-CC specifies and experiments vehicular communication, and promotes the harmonization of vehicular communication standards worldwide.

Regarding the embedded applications design, works have been done around Java. The OSGi alliance promotes an horizontal framework for the deployment of distributed applications. It is based on Java and relies on the concept of bundles that extends the concept of class. OSGi implements a dynamic service-oriented programming and includes the concept of component. The car manufacturers have been contributed to the OSGi and an extension has been proposed to manage the sensors inside a vehicle [14].

Currently, not all the projects have designed their embedded architectures, nor released related public documents. The GST architecture is based on Linux, OSGi, IPv6, HTTP, SOAP and OMA DM protocols [6]. The CVIS project architecture relies on Linux, OSGi, IPv6 and CALM [3].

1.4 Contribution

The envisaged architectures often rely on existing standards. This reduces the development time by reusing software and simplifies the portability and deployment issues. Hence, existing standards would help to the launch of the new ITS business.

However, not all the existing standards may be adapted to the vehicular networks. Such networks are highly dynamic, unreliable and asynchronous. They are generally penalized by a low bandwidth and a short communication duration. Despite these difficulties, ITS applications require often robustness, quality of services or real time management. Last but not least, still many research issues remain open and the embedded architectures should allow the integration of future protocols, distributed algorithms and programming paradigms.

It is our point of view that highly dynamic networks are not like any other computer networks where already known solutions can be deployed. Such networks are still largely unknown and the standard required for deploying distributed applications should be light and open to any new solution. With the increasing number of equipped mobile nodes, future yet unknown services and usages will certainly appear, with their related problems. The embedded distributed framework should only fix the overall architecture and a very light and open data exchange protocol, in order to remain open to any protocol stack, programming paradigm, and future improvement, while still allowing new services deployment.

In Section 2, the requirements of the embedded architectures are discussed on the basis of the ITS open issues. In Section 3, a simple architecture – named Airplug – is proposed, which fulfills the requirements. It focuses on the data exchange scheme and remains open to any addressing scheme, protocol stack and programming language. Sections 4 details an implementation of the Airplug architecture in a light framework, and reports experiments. Concluding remarks end the paper in Section 5

2. ARCHITECTURE REQUIREMENTS

In this section, we analyze the requirements of embedded distributed frameworks for vehicular networks. We begin by sketching the open issues.

2.1 ITS open issues

One of the main problem of ITS applications concerns the dynamic of the network. Inside a given perimeter, the arrival and departure rate of vehicles is variable and can be very high. The vehicles density is sometimes high, sometimes low. Moreover, many obstacles and interferences can disturb the signal propagation and the quality of reception (eg. trucks, buildings...). This leads to difficulties for the access medium layer, which has to fairly share the bandwidth between vehicles. Research studies are currently ongoing for the link layer to ensure fairness, efficient global throughput and quality of service.

It is important to note that cellular-like networks cannot solve all the ITS issues. For instance, if such an infrastructure-based network can be used to broadcast a message in a given area, it is much more difficult to use it for warning only the vehicles which will encounter a given road accident. To the contrary, solutions have been proposed to solve this problem in the inter-vehicles networks [22, 18], so-called Vehicular Ad hoc NETwork (VANET). A VANET is a special case of MANET (Mobile Ad hoc NETwork) and routing protocols studied for ad hoc networks may be used for ITS applications. However, as the neighborhood of any vehicle and the whole topology are very unstable, the classical MANET protocols are not always efficient. Proactive algorithms consume bandwidth to build unstable routing tables; reactive algorithms experience many route breaking; GPS-based routing needs to address a large area to reach a fast mobile vehicle [18]. Research studies are currently ongoing for V2V and V2I routing.

The network dynamic is also a difficulty for data flow transportation. Routes are unstable and wireless communications error prone. The bandwidth estimation is then very approximate and the optimal sending rate is variable and almost impossible to determinate. Transport protocols such as TCP often experience problems in wireless networks because they interpret layer two collisions as more durable routing problems (congestion in layer 3). Hence, using the Internet stack including TCP, IP, HTTP, SOAP... in the vehicles may lead to some performance issues especially in low bandwidth networks. For instance, in many cases Internet addresses can be replaced by node's ID, which in turns are not always mandatory in such opportunistic networks [18]. Hence, besides the routing algorithms, research are still required on the protocol stack.

Another important issue is the construction of robust distributed applications over such a network. When interacting each others, the embedded programs compose a distributed application, running on an unreliable asynchronous network where nodes may disappear. However several important issues are still unresolved in such networks. It has been proved that consensus cannot be solved [19] while taking a common decision is a key issue for many applications: braking of several cars with no multiple crash, insertion of a vehicle (cooperative driving), distributed games... What we could name *best effort algorithms* may replace ideal but unreachable deterministic algorithms, with consequences on the use of the results and, finally, on the usages.

Many ITS applications are sensible to attacks: safety applications require protections to ensure their service; value added applications need protections face to intruders; traffic forecast and diversion services have to be secured against false information... However, securing such a network is not easy [24]. Certificates based confidence systems could be adapted. But such so-called VPKI (Vehicular Public Key Infrastructure) require a revocation key system, which cannot be efficiently implemented in a network with episodic connections to an authoritative server. Moreover, well certified messages may contain false information.

Many other challenges ought to be invoked, such as privacy preserving, accurate car positioning, dynamic maps (adding dynamic attribute to embedded maps), real time issues, context aware optimization...

2.2 Requirements analysis

Besides the technical issues, the success of the ITS applications depends on some business issues such as the consumer acceptance or the attraction of the business model. To launch this new activity, the profitability of the new services should be ensured. It depends on the number of equipped cars, on the ease of deployment of new applications, on the efficiency of these new services... The embedded architecture is then crucial.

Such an architecture should be cheap to be rapidly integrated in a large number of vehicles. It should ease the deployment of new applications in the vehicles. Note however that ensuring an easy deployment should not impact the programming language nor paradigm. As continuous improvements appear in software programming, the ITS standard should focus on the applications architecture and not on their implementation.

The architecture should also remain open to any new technology, protocol or algorithms. Indeed as we noticed, many scientific challenges are studied by research teams and new efficient solutions could appear while the framework is already installed in many vehicles. Moreover several concurrent implementations of the architecture should be possible in different but interoperable frameworks. Hence, the standard should be minimal and should focus on the exchange protocol.

To increase the user acceptance, the services should be reliable. This implies robust and light frameworks, as well as a clear independence between the embedded applications (to

limit the error propagation). Indeed, when a task fails or has a hieratic behavior, the rest of the system must continue to work. The more the tasks are independent, the more the robustness can be ensured. Moreover, the architecture should be defined on top of a robust operating system (eg. with user spaces applications) and the resources allocation, tasks scheduling and real time management should be delegated to it. This is important to avoid any redundancy with the classical operating systems and to allow to take benefit of the continuous improvements in operating systems developments.

Finally, the architecture should allow efficient framework implementations to offer good performances. Complex protocols stacks that lead to latency, overheads and large headers should be avoided when possible.

3. THE AIRPLUG ARCHITECTURE

In this section, we sketch a software architecture – named Airplug – that fulfills the above requirements. It is dedicated to the design of distributed applications in highly dynamic ad hoc networks, such as the vehicular networks.

3.1 Airplug architecture design

Process-based architecture. In order to obtain a light, portable and robust distributed framework, the Airplug architecture relies on the facilities given by standard operating systems: resources allocation, process scheduling, real-time management... This avoids any redundancy between the framework and the operating system, and allows to take benefit of any improvement in these fields (including real time management). It is expected a portable POSIX operating system with process management and memory protection for robustness issue.

The architecture has been designed to remain open to any future solutions. Hence it imposes very few common conventions to the applications, which can be developed with any programming language. The applications do not need to include network operations nor complex inter-process communication primitives. By this way, the applications do not depend on the framework and can be used standalone. This permits any future improvement either in the programming paradigms, the network protocols or the framework itself.

The framework is concentrated in a single core program – called *airplug* – per mobile node, that runs in a standard process on top of the operating system. By not including any part of the framework in the kernel nor in the applications, the independence with the operating system as well as with the applications programming is enforced.

The Airplug architecture accepts either local or distributed applications. A local application does not have any interaction with remote applications. A distributed application is composed of several instances of the same program, running in different mobile nodes, and exchanging messages. Local applications as well as local instances of the distributed applications run in separate processes with their own memory space. This enforces the applications independence as well as the reliability: an application may be bogus with very few impact on the rest of the system. Moreover, this allows to subcontract the scheduling of the applications as well as the real-time management: the framework should just set the priorities regarding parametric rules (eg. context-aware heuristics) and the operating system does the rest.

All these processes are launched by *airplug*, which creates descendant processes, called *plugged processes*. By this way, *airplug* is easily informed of the problems of an application by catching the related signals sent by the OS to the parent process (eg. abnormal termination).

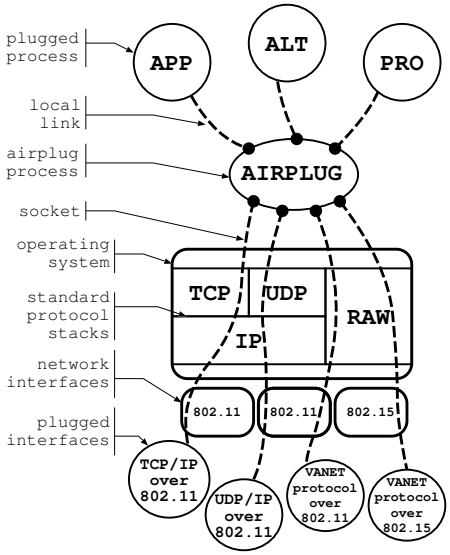

Figure 1: A node in the airplug architecture.

Inter-process communications. To fit with the asynchronous opportunistic network, the protocol is based on asynchronous messages passing. In order to minimize the requirements for the applications development, inter-process communications are done with messages through standard IO. Indeed, any process owns by default a standard input `stdin`, a standard output `stdout` (and a standard error output `stderr`). This functionality is then supported by any programming language, and it gives no requirement on the application programming. The standard IO are sufficient to perform the inter-process local communications, and it can easily be extended to ensure inter-process remote communications (between distant nodes). Moreover, such a communication scheme allows easy standalone use of the applications (without *airplug*) and it permits to reuse existing applications.

For each *plugged process*, standard input and output are redirected from and to *airplug* via bi-directional connected communication links called in the following *local links* (see Figure 1). There is one *local link* per *plugged process*. By this way, each time a *plugged process* writes on its standard output, *airplug* will receive the data via the related *local link*. And each time *airplug* writes on a *local link*, the related *plugged process* will receive the data. The *airplug* program scrutinizes the *local links* to receive the data from the *plugged process* and forward them to the destinations (local or distant process specified by the sender).

Hence *airplug* represents a kind of "bus" between all the *plugged processes*. The network interfaces are also connected to this bus so that any application on the top of an *airplug* instance in the vicinity are connected to a common applicative bus. This bus can be extended by means of multi-hops communications. It is important to note that this bus is very simple and can then be efficiently implemented by avoiding any kind of unwanted processes synchronization. This simplicity is well adapted to the opportunistic networks: it provides quick communication with very few common conventions and without any global management (eg. services directory); it then supports rapid extension or reduction depending on the nodes movements.

Language independence. Besides the efficiency, this communication scheme preserves the language independence. Even shell scripts programs can be used. To the contrary, more sophisticated inter-process schemes (shared memory, MPI, CORBA, Java RMI, Web services, OSGi...) are restrictive and limit the language choice or requires specific libraries. Here, all the communication stuff is implemented in *airplug* and applications just have to read and write to their standard IO.

The only constraint is for programs which have at least a second entry such as the keyboard (for interactive applications), a camera... These applications should then be able to read several entries. This can be done for instance by asynchronous reception. Whenever the operating system signals that an entry can be read, the process checks its entries.

Thanks to the language independence, the more adapted programming paradigm can be chosen to build an *airplug* compatible application: object oriented or not, multi-threaded or not, interpreted or compiled... Other criteria can be considered such as the experience of the programmers, the code reusing, the compatibility with other environments (see Section 4)... Reading and writing on standard IO is very general, and many existing applications can be used. A small adaptor can format the output of such existing applications to conform to the *airplug* conventions. This can be implemented as a standalone program inserted between *airplug* and the legacy application (for instance through a pipe `|` with a shell script).

Finally, the language independence allows to take benefit of any language and compiler improvement, and to remain open to new future programming paradigm. Assuming a similar embedded computers architecture in each vehicle, the applications could be deployed using the operating systems packages management tools.

Networking integration. As explained earlier, a layered structure with the framework on top of an operating system is advantageous to avoid any redundancy. However the optimal structure is currently not really clear regarding networking because many research works investigate so-called cross-layering solutions. Hence, it appears necessary to have a versatile architecture instead of a strict layered structure between the network and the applications, in order to authorize any by-pass.

In the Airplug architecture, the network interfaces are accessed through *airplug*, and are called *plugged interfaces* (Figure 1). The *plugged interfaces* are managed as the *plugged processes*, to the exception that they are connected to *airplug* via some sockets. Hence, the network is addressed by the applications in the same way they address a message to another application, simply by writing to their standard output. The *airplug* program receives the data sent by the *plugged processes* and sends them to the desired *plugged interface* through the related socket.

By using several *plugged interfaces*, several network devices can simultaneously be accessed, an several protocols can be used. For instance, sockets may be open for the Bluetooth interface and other to some WiFi interfaces. Some of them could use the TCP/IP stack while some other could perform broadcast over UDP. By allowing to use the network stacks included in the kernels, *airplug* ensures the compatibility with any existing network such a Internet, and can take benefit of any improvement of such protocols. For instance, IPv6 may be used by including it in the kernel, and by opening the corresponding socket.

But this architecture is also open to any new communication scheme. By using `raw` sockets, the link layer can directly be accessed through a *plugged interface*. This can advantageous be used to broadcast in the neighborhood by avoiding the IP and UDP headers. Moreover, this allows to implement in user space new protocols inside *plugged processes*. Such a process will receive the messages from the applications that want to send their data using the new protocol. It will then send messages through a `raw` socket to the remote hosts (Figure 2). Hence, any by-pass of the Internet stack as well as cross-layering solution can be designed by simply using the inter-applications facilities of *airplug*.

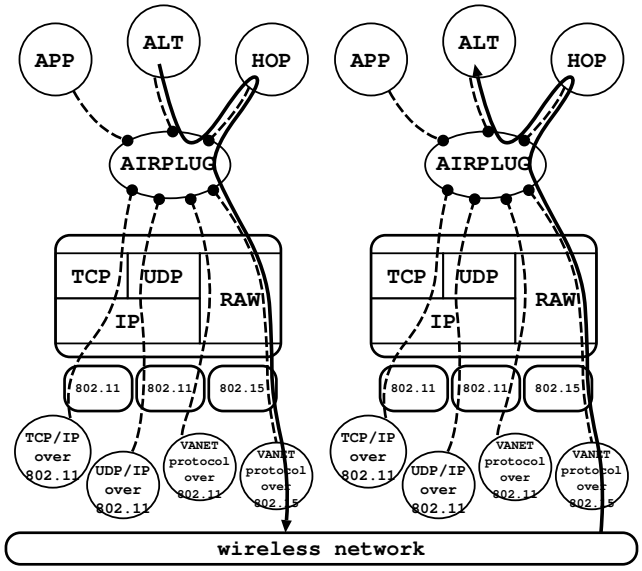

Figure 2: Integration of a new protocol in the airplug architecture. Here the HOP protocol is used by the ALT application.

3.2 Airplug protocol

Protocol philosophy. By Airplug protocol, we mean the common conventions necessary to the inter-applications communications. As previously explained, a local application runs in a process launched by *airplug*. Any service (such as a network service) is implemented in such a process. A distributed application is composed by several instances of the same local application, running on different hosts. On a given host, an application may be composed of several processes if some parts of this composite application may be addressed by other applications directly.

Hence, a process represents the addressable unit in the Airplug architecture and any communication is performed between two processes. This implies to be able to address a process.

Addressing. Locally, a process is designed by the name of the application. This name is used by *airplug* to find the concerned *local link*. Locally, the application names must be unique. The keyword `ALL` designates all the local applications.

Since the Airplug architecture can integrate any protocol, it does not require IP. Hence the host names cannot rely on IPv6 addresses. In fact, designing a host by a network address is not efficient in a highly mobile network [18] because this is not a durable information. Moreover, working with the neighbor's name (ID, pseudonym...) could be costly when the neighborhoods are very unstable.

Based on these considerations, the Airplug architecture implements a very simple addressing scheme, well adapted to opportunistic networks. A message can be sent to an application i) on the local host (`LCH`), ii) on the other hosts in the vicinity by means of wireless interfaces (`AIR`) or iii) on both (`ALL`).

A fourth mode can be used when a discovering service allows to detect the names of the neighbor hosts. In this case, a neighbor name can be used instead of one of the three keywords `LCH`, `AIR`, `ALL`. If an instance of *airplug* receives a message addressed to another node, it will discard the message. Note however that many applications do not require such a service. Moreover the conditional transmissions [18] allow to designate the relay and destination nodes by means of conditions (eg. nodes behind the sender). This multi-hop communication strategy is well adapted to dynamic networks: it allows to perform one-to-many communications (the most used communication scheme [20] in VANET network) without knowing the identity of the nodes.

To conclude, a process is addressed by a couple `(APP,HST)`, where `APP` is either the name of an application or the keywords `ALL` or `CTL` (used in certain cases, see under) and `HST` is either the keywords `LCH`, `AIR`, `ALL`, or the name of a neighbor node.

Communication primitives. With this addressing, a message can be addressed to one or several applications locally or remotely through a single primitive denoted by `SND` (for "send"). However an application cannot receive the messages addressed to several applications nor received by `AIR` until it has notified *airplug* for such a subscription. An applications can then control its receptions. This increases the robustness by avoiding any cascading problem in case of bogus application. In practice, this allows to test and debug new applications while simultaneously performing useful experiments with already tested applications. The subscriptions are notified to *airplug* with the `BEG` and `END` actions (for respectively the "beginning" and the "end"). The *airplug* program manages the applications subscriptions through some lists. An application can subscribe to a specific application or to all (`ALL`), either locally (`LCH`) or remotely (`AIR`). It is also possible to subscribe to any control information (`CTL`).

With such a communication scheme, it is possible to implement the pull mode (request/answer) as well as some push modes. Indeed, using the `ALL` keyword, an application will send some messages to all the applications that subscribed to its messages. Such sending can be done either periodically (eg. GPS program) or when an event occurs such as the change of a value (eg. visibility distance). The sending mode described in the `wireadmin` OSGi package can then be implemented with *airplug* (sufficient variation, threshold, hysteresis...).

Message formats. Basically, a message is composed of the action field, the address field, the control field and the payload field. However the message format depends on the kind of communication, either local or remote (Figures 3 and 4).

The action field can contain `SND`, `BEG` or `END`. Since the subscriptions concern only local communication, the action field is only present in local messages.

The address field contains one address `(APP,HST)` for the local communications and two for the remote communications. Indeed, a distant communication concerns several instances of the *airplug* program, and is done on the network through a socket. Since such a communication is generally performed by broadcasting in the neighborhood, the message contains the addresses of the sending and the receiving applications. To the contrary, a local communication concerns *airplug* and one of its son processes. It is done by writing and reading on a local link, which is a connected communication link. In this case, a single couple `(APP,HST)` is sufficient because the sender is known by the receiver. For a message sent on the local link by an application, it is used to describe the destination (sending action `SND`) or the subscribed application (subscribing action). For a message sent on the local link by *airplug*, it contains the sending application.

The control field is used for piggybacking purpose, to send some optional data with the messages such as identity, geographic position etc. Any application as well as *airplug* itself can be interested by the data in this field, to the contrary of the payload field. The applications can subscribe to the control data by using the `CTL` keyword instead of the name of an application.

The payload field is application dependent and contain the data really exchanged between the processes.

action	application	host	control	payload
SND		LCH		
BEG		AIR		
END		ALL		
		hostname		

Figure 3: Format of a message for a local communication through a *local link*.

appl.	host	appl.	host	control	payload
sending	sending		AIR		
appl.	hostname		ALL		
	(if any)		hostname		

Figure 4: Format of a message for a remote communication between two hosts through sockets.

The messages are sent in plain text format. This allows human reading and eases the debugging. Moreover this allows to use the applications in standalone mode (in this case, the applications read an write from the terminal). When necessary, the payload can be encoded using an efficient algorithm such as [15]. For optimization purpose, an optimized mode allows the transmission of binary payloads.

The different fields are separated by a character given at the beginning of the message; this character should not be present in the fields except the last, which is application-dependent (payload). This permits to use variable length fields, which is well adapted to the Airplug named-based addressing scheme `(APP,HST)`. For optimization purpose, an optimized mode allows to use a prefix of the names instead of the complete names (the length of the prefix has to be precised regarding the number of applications).

4. EVALUATION OF THE ARCHITECTURE

4.1 Implementation

In order to evaluate the Airplug architecture on real scenarios, a prototype of the core program as well as a set of applications has been implemented. The core program has been written in C under Linux but is intended to work on any POSIX operating systems. A careful attention has been paid on the robustness. The program – named `apg` – relies only on the standard `libc` library and is compiled with `gcc`. The executable is less than 40 Ko. The source code has less than 3300 lines for 177 Ko. This shows the lightness and the portability of the framework.

The *plugged processes* to be created and the *plugged interfaces* to be configured have to be given as arguments of `apg`. The *plugged processes* are launched using `fork` and `exec`. The local links are implemented with `pipes`. Inter-applications exchanges are done with messages-based asynchronous communications. They require two `write` and two `read` for local communications (through `pipes`) and one more `write` and `read` for remote communications (through `sockets`).

With the Airplug architecture, any language can be used to write the applications. Thanks to this characteristic, we developed most of our applications with Tcl/Tk in order to share code with Network Simulator [23]. These applications have been designed with an event-oriented programming (ie. actions are done only when an event occurs), which is well adapted to distributed applications. The standard input is configured in such a way that a reception on `stdin` (data sent by `apg`) is an event. Among the tested applications, we may quote: GPS reading, camera reading, neighborhood discovering, file transfer, instant messaging, distributed games, road visibility foreseeing, convoy detection, traffic road characterization, alert diffusion, conditional transmissions, VANET optimized broadcasting, streaming transfer, end-to-end delay and bandwidth measurement for multi-hops communications...

The discovering service can run either standalone or by piggybacking.

The conditional transmissions are an example of VANET specific protocol, which has been implemented in user space in a *plugged process* named `HOP` (Figure 2). A message is sent with two conditions, CUP and CFW for upward and forward respectively. At the reception, the message is transmit to the local application if CUP is true and it is forwarded to the neighbors if CFW is true. The CUP condition defines the destination nodes and the CFW condition defines the relay nodes. By evaluating the conditions at the message reception, this communication strategy appears much more adapted to dynamic networks than classical ones [18]. When an application want to use this routing scheme, it addresses its message and the CUP and CFW conditions to the `HOP` application through `apg`. Then `HOP` will add some control data before sending the messages to the neighbors via `apg`. Reciprocally, when `apg` receives a message addressed to `HOP` from a network interface, it transmits it to the local `HOP` instance, which will decide whether it will resend the message to the network and/or to a local application (depending on the CUP and CFW conditions).

We used yEnc encodage [15] for streaming transfers in plain text messages, with an overhead of 3%.

4.2 How to design applications

Data flow. While the Airplug architecture allows many more varieties of applications construction, we basically distinguish four classes of applications, sorted by their data flow.

The application of the first class are the simplest. They produce a local output only, on the basis of a local program, a local file or a local device (sensor, camera, GPS...). They generally broadcast their output locally (`ALL,LCH`). Examples are GPS reading and image capture... The second class of applications consume local data produced by the applications of the first or second class. They are used to perform some computations on data produced by class one applications, or for aggregating these data. They are also used to produce more long time analysis by storing the data during a given period (eg. averages). They send their data locally, either on request or by broadcast. Examples are traffic road characterization, visibility distance computation... The third class of applications is a special case of class two applications which collect or send some data by air. However they do not implement a distributed algorithm. An example is the neighborhood discovering service, file transfer,

road visibility foreseeing... Finally, the fourth class of applications is composed of distributed applications. They are built with one instance per host which cooperate to achieve a global goal, that defines the distributed applications. Examples are convoy detections, instant messaging, distributed games...

Applications that lead to some loops in their data flow should be designed carefully to ensure their robustness and stability. Distributed applications should be designed with attention: proving their correctness when running in a VANET – a kind of asynchronous failure-prone wireless network – may be hard.

Example. Let suppose that a discovering service `DVS` has to be implemented on the top of `apg`. It is composed of one instance of `DVS` per mobile nodes, that send periodically some `beacon` messages (Figure 5).

To receive the data from the GPS, on each host, the `DVS` application subscribes to the local `GPS` applications by writing to their standard output the message `_BEG_GPS_LCH_-_-`. Here, the separator field character is `_`. To receive the `beacon` messages from the remote instances of `DVS`, it subscribes by writing `_BEG_DVS_AIR_-_` to `stdin`. These messages are received and interpreted by the local `apg` program. On each host, the `GPS` application pushes its data every second by writing the messages `_SND_ALL_LCH_-_gpsdata` to its standard output. When receiving such messages, the local `apg` programs write the message `_SND_GPS_LCH_-_gpsdata` to the `pipes` of the subscribed local applications (including `DVS`). To send the `beacon` messages, a `DVS` application writes the message `_SND_DVS_AIR_-_beacondata` to its standard output. The local `apg` program will then broadcast the message `_DVS_hostname_DVS_AIR_-_beacondata` in the neighborhood. All `apg` programs in the vicinity will receive this message and will write the message `_SND_DVS_AIR_-_beacondata` to the subscribed application (including the local `DVS` instances). Hence, the `DVS` application can exchange some data that include the local GPS positions. When another application requires the bandwidth, these data can be sent in the control field of the messages (piggybacking). Hence, the `DVS` application should also subscribe to the `CTL` field by writing `_BEG_CTL_AIR_-_` on its standard output.

Note that, thanks to the standard input/output based communication scheme, the `DVS` application can simply be tested standalone by typing on the command line: `DVS | DVS`.

4.3 Experiments

Our implementation of the Airplug architecture has been designed for the experimental study of highly dynamic networks such as VANETs. We would like to test on the road our theoretical results (routing, embedded distributed algorithms...), as well as to guide our further studies by inputs from experiments. For this purpose, a simple embedded platform has been used [17]; it is composed of industrial PCs, GPS and WiFi antenna (Figure 6). The operating system is Debian 3.1.

We performed various tests on the road with up to six vehicles. The different applications were able to run simultaneously. The variety of these applications shows that the Airplug architecture allows to design a very large set of ITS applications.

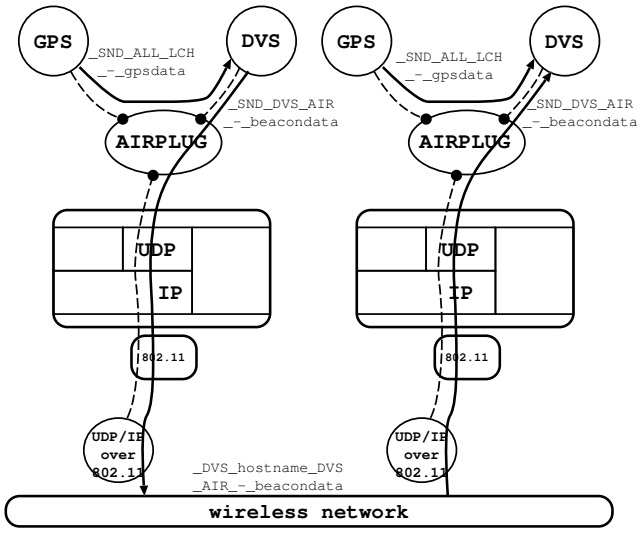

Figure 5: Messages exchanged for the discovering service `DVS`.

Figure 6: Embedded hardware for the Airplug architecture evaluation.

5. CONCLUSION

Among all the ITS open questions, the embedded software architectures specification is currently a hot topic, and one of the crucial open issues. Using existing frameworks and protocols may ease the launch of the ITS applications business. But not all of them may fit with the applications prerequisite on highly dynamic networks.

In this paper, we analyzed the software architecture requirements. Due to the open research issues, a light open architecture seems necessary in order to be able to adapt to any further improvement, solution or... problem that may arise in such yet unknown networks. Standardizing to much features may indeed be a drawback in this new field.

We then defined a light software architecture called Airplug. It allows to implement all the framework stuff in a single process. This ensures the independence with the operating system and avoid any programming constraints on the application development. Resource management and tasks scheduling (including real time management) are delegated to the operating system. Applications are implemented in separate processes to increase robustness. Inter-process communication relies only on standard input and output. By this way, applications can be written with any language. Moreover already developed applications can easily be integrated. Existing protocols stacks can be used but the integration of new protocols is facilitated. While remaining compatible with existing addressing scheme, the architecture does not require IP and proposes a new addressing scheme well adapted to opportunistic networks.

In order to evaluate this software architecture, we implemented on Linux. This leads to a very light framework. A large set of applications have been developed on top of this framework. They have been tested all together during real experiment on the road. These tests demonstrate the interest of the Airplug architecture. As a conclusion, we think that ITS applications could take benefit of a light and efficient framework.

We plan to complete this study by measuring the sending latency at the application level and the CPU load when many applications run... Comparisons with other frameworks would be interesting. Note that since the framework is light, good performances are expected. We showed here that the lightness of the framework is not a drawback for writing complex application.

Airplug is currently used for studies regarding the context aware optimizations by dynamically reordering the local links. Future works concern the study of distributed algorithms over dynamic networks and their evaluation in real situations. An emulation mode is under development. It will allow to replay the real experiences in laboratory.

Aknowledgement. The author wishes to thank Mohamed Shawky for useful contributions, as well as Khaled Chaaban, Gaël Delbary, Nicolas Eude, Yacine Khaled, Sofiane Khalfallah, Stéphane Pomportes and others for the development of Airplug compatible applications.

6. REFERENCES

[1] The Cooperative Intersection Collision Avoidance (CICAS) initiative. http://www.its.dot.gov/cicas/.
[2] The COOPERS project. http://www.coopers-ip.eu/.
[3] CVIS making the connection. http://cvis.odeum.com/download/cvis_first_brochure.pdf.
[4] The CVIS project. http://www.cvisproject.org/.
[5] The eSafety initiative. http://www.esafetysupport.org/.
[6] GST open systems architecture and interface specification. http://www.gstproject.org/os/documents/DEL_GST_OS_DEV_Reference_Implementation_3_2.pdf.
[7] The GST project. http://www.gstforum.org/.
[8] The Intelligent Transportation System. http://www.its.dot.gov/its_overview.htm.
[9] The national ITS architecture. http://www.its.dot.gov/arch/index.htm.
[10] The Open Mobile Alliance (OMA). http://www.openmobilealliance.org/.
[11] The PREVENT project. http://www.prevent-ip.org/.
[12] The SAFESPOT project. http://www.safespot-eu.org/.
[13] The Vehicle Infrastructure Integration (VII) coalition. http://www.vehicle-infrastructure.org.
[14] Wire admin service, OSGi package `org.osgi.service.wireadmin`.
[15] yEnc: efficient encoding for usenet and email. http://www.yEnc.org.
[16] The Car-to-Car Communication Consortium (C2C-CC). http://www.car-to-car.org.
[17] The Caremba platform. http://www.hds.utc.fr/caremba.
[18] B. Ducourthial, Y. Khaled, and M. Shawky. Conditional transmissions, a strategy for highly dynamic vehicular ad hoc network. In *IEEE WoWMoM'07*, 2007.
[19] M. Fisher, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):373–382, 1985.
[20] Y. Khaled, B. Ducourthial, and M. Shawky. A taxonomy for routing protocols in vehicular ad hoc networks. In *Workshop UbiRoads of IEEE GIIS'07*, 2007.
[21] K. Lee, S.-H. Lee, R. Cheung, U. Lee, and M. Gerla. First experience with cartorrent in a real vehicular ad hoc network testbed. In ACM, editor, *VANET MOVE'07*, Anchorage, Alaska, May 2007.
[22] C. Maihfer. A survey of geocast routing protocols. *IEEE Communications Surveys and Tutorials*, 6, 2nd quarter 2004.
[23] Network simulator. http://www.isi.edu/nsnam/ns.
[24] M. Raya and J.-P. Hubaux. Securing vehicular ad hoc networks. *Journal of Computer Security*, 15(1):39–68, 2007.