Designing applications in dynamic networks: The Airplug Software Distribution

Bertrand Ducourthial

Designing applications in dynamic networks: The Airplug Software Distribution
(Invited Paper)

Bertrand Ducourthial*

UMR CNRS 7253 Heudiasyc,
Université de Technologie de Compiègne
BP 20529, 60205 Compigne Cedex, France
Bertrand.Ducourthial@utc.fr

Abstract. In a dynamic network, a communication link can only be used for sending very few messages before disappearing. This happens whenever the nodes mobility is high with respect to the underlying communication protocol.

Starting from these conditions, we propose design rules for distributed applications, which arises implementation rules. We then introduce the simple yet powerful Airplug framework.

The Airplug Software Distribution includes several implementations conformed to the Airplug framework for prototyping, real tests or studies by emulation. It is a convenient set of tools for designing complex applications for dynamic networks.

This approach has been validated by many applications and experiments, mainly in the field of vehicular networks.

1 Introduction

The democratization of communicating terminals has led to new uses. As they are more and more small and inexpensive, they are integrated into probes (wireless sensors), cars (connected vehicles), objects (Internet of Things) or robots. Communication can rely on an infrastructure network, as for instance for the so-called *connected vehicles* which uses 3G mobile operators to reach Internet. Nevertheless, relying on an infrastructure network is not always convenient:

- As infrastructure networks are expensive to deploy and to exploit, a subscription is required.
- Large range communication are required to reach an infrastructure access point. Consequently, many communicating nodes compete in the same geographic area, limiting the bandwidth.
- Infrastructure networks do not offer infinite capacity while the demand will always increase.

* This work was partially carried out in the framework of the Labex MS2T, which was funded by the French Government, through the program " Investments for the future managed by the National Agency for Research (Reference ANR-11-IDEX-0004-02)

- Infrastructure networks cause a delay that can be detrimental to the system responsiveness. For example, an emergency braking alert system on highway should preferably rely on inter-vehicle communication instead of operating system [12].
- By emitting at large range to reach an access point, more power is required. This can be a drawback for the embedded battery and/or for the user.

Depending on the context, systems are more or less impacted by all these criteria. We assist then to the development of autonomous networks that connect the communicating devices themselves. Specific protocols are defined: ZigBee (IEEE 802.15.4) for sensors, WAVE/802.11p for vehicles, Bluetooth improvement (IEEE 802.15.3) for personal devices... New research area emerged, such as WSN (Wireless Sensor Networks), VANET (Vehicular Ad hoc Network) or cooperative robotics with fleets of robots or drones (Swarm robotics).

From an application point of view, we leave the centralized control offered by the network infrastructure to a distributed control: instead of gathering information on a server that calculates and returns its answer, calculations are done in the network by the communicating devices themselves. This leads to distributed applications with the specificity that the underlying network is highly dynamic.

At this step, we need to define what we intend by *dynamic networks*. Roughly speaking a dynamic network is a network where stability does not exist for a long time. However assuming that any communicating link disappears after δ seconds does not give necessarily a dynamic network: this depends on the amount of data the communicating protocol is able to forward during δ seconds. In [7], we proposed a new metric related to the number of messages that can be sent over a link before it disappears. Reporting work in progress in this field is out of the scope of this paper. In the following, we assume that, in a dynamic network, only few messages can be sent over a link before it disappears.

Assuming this constraint, how designing distributed applications for such networks? Many attentions has been given on communicating protocols (e.g., IEEE 802.11p for vehicles), few for the new challenges in distributed algorithms, still less on the implementation. Our paper deals with this last point. We begin by sketching design rules for applications. Then, we analyze what a framework should offer to implement distributed applications in such a context. Finally we present our solution, namely the Airplug Software Distribution.

2 Design rules

In this section, we give several design rules for applications dedicated to dynamic networks. These reflections will guide the implementation.

Topology. As the links only allow sending few messages before disappearing, the topology is unstable. Hence a distributed algorithm should not assume any characteristic regarding the topology. In some cases, distributed algorithms assume the existence of a virtual structure on which they rely, such as spanning trees or clusters of nodes.

However, maintaining such a virtual structure requires some control overheads, which consumes the bandwidth. In a dynamic network, the structure would never be usable while consuming too much messages. Except in some particular cases (e.g., regular convoy of vehicles), virtual structures should be avoided.

Note that several algorithms designed for MANET (Mobile Ad hoc Networks) assume an acyclic underlying network (mainly to avoid any loop), requiring then a spanning tree. This is certainly a strong difference with dynamic networks.

Note also that algorithms assuming particular topology patterns to run properly need to monitor the network to check whether the pattern appears or not.

Non-local knowledge. Some algorithms may require at some point in their execution a non-local information, either qualitative (e.g., the presence of a specific node such as a gateway) or quantitative (e.g., the distance to a specific node).

Obtaining such a remote information requires to send at least one request by means of multi-hop communication and then to obtain the answer, still by using multi-hop communication. However, as links disappear rapidly, there is no guarantee that a return path will exist. Moreover, when the answer reaches the initial node, it could be false because the remote situation would have change. For instance, the gateway node is no more in the same connected component of the dynamic network. We address this problem in [11]. The same problem arises with any knowledge requiring remote information, including obviously global information: a distributed application on dynamic networks should avoid relying on varying non-local information.

Addresses. A network address includes generally the name of the node as well as its position in a given frame of reference such that it points to a unique node. Addresses are used for routing messages. Most common frames of reference are the topology (topology based routing protocols) or the geographic map (georouting).

However in a dynamic network, the node's positions vary and this part of the address is never stable. We address this problem in [10, 11]. When addressing a message to a remote node, the senders must know its address. This implies that it must know its position in the network, which is a non-local information. This is generally done using a location service [16], which requires much communication for searching nodes. Such a search in a dynamic network would consume too many resources and return an outdated information.

Hence, while nodes identifiers can still be considered, network addresses, to the contrary, should not be used in dynamic networks.

Neighborhood discovery. When routing a message, the next hop can be decided either on the sender side or on the receiver side. We address this problem in [8].

In a sender side scheme, each time a node receives a message to be forwarded, it has to determine which node will be the next hop. To the contrary, in the receiver side scheme, the node broadcasts the message in its vicinity and nodes receiving it have to determine whether they retransmit it or not.

Note that wireless communication relies on local broadcast. With the receiver side scheme, the decision is taken later than in the sender side scheme. However several nodes could decide simultaneously to retransmit the message. But this can efficiently be solved with contention algorithms.

The sender side scheme needs to learn about the neighbors and their characteristics (e.g., geographic position, move direction...). This requires periodic communication which consumes the bandwidth. Moreover, the selected relay node could disappear before receiving the message to be forwarded or its characteristics may have change so that it is no more the good choice.

Hence, as the neighborhood is unstable, it is preferable to rely on receiver side scheme in dynamic networks.

Conclusion. When communication links allow to send only few messages before disappearing, it is vain to rely on any non local knowledge. Remote node position or neighborhood are example of such non local knowledge. The more the dynamic increases, the more the cost (in messages) increases while the preciseness decreases. This can be summarized as follows:

$$\boxed{\text{Dynamic} \nearrow \Rightarrow \begin{array}{l}\text{cost} \nearrow \\ \text{preciseness} \searrow\end{array} \Rightarrow \text{usefulness} \searrow}$$

3 Implementation requirements

In this section, we give requirements for a programming framework dedicated to dynamic networks.

Hardware, OS and language agnosticism. Dynamic networks appear each time the communication protocol and the nodes mobility leads to the fact that only few messages can be sent over a link before it disappears. This is a general definition accepting many applications: vehicular networks, wireless sensor networks, drone or robot networks... The programming environment should then be usable on a large variety of communicating terminals; it should be hardware agnostic.

Moreover, it should also be language independent because the variety of situations will be better tackled if specific languages can be used.

Similarly, embedded devices often use specific operating systems; the less there are some constraints on them, the more the programming environment will be deployed. Moreover, still to increase the portability, the framework should not require any modification of the kernel itself.

Note that a framework that remains independent from any hardware, OS and even language requirement is able to take benefit of any novelty or improvement in these third fields.

Process-based architecture. In a process-based architecture, each task is implemented in an independent process. Designing a programming framework using a process-based architecture offers several interesting properties, compared for instance to a single multi-threaded process. Efficiency is not affected as long as there are not many tasks creations during the execution. This can be done by launching all the tasks at the beginning of the execution, even if some of them are sleeping.

Modern operating systems offer many tools for controlling processes, to the contrary of (user) threads. Resource allocation, tasks scheduling and real time management can be delegated to the OS. This allows to avoid any redundancy with OS and to take benefit of the continuous improvements in operating systems developments.

When installing several third party applications, robustness will be better ensured if each of them is implemented in independent processes. Indeed, when a task fails or has a hieratic behavior, the rest of the system will continue to work. The more the tasks are independent, the more the robustness can be ensured.

Communication-based conventions. The framework should rely on a reduce set of conventions to ease the portability and the implementation of several concurrent but interoperable frameworks. A solution consists in focusing on message format only, limiting requirement for programs themselves. This will lead to light frameworks; robustness will be better ensured and verified.

Regarding the communications, they will appear between local and remote processes. When two local tasks exchange messages, any inter-process communication (IPC) tool could be used, providing it is available on any language and any OS.

Communication between remote tasks should not depend on a specific communication protocol. Moreover, as explained above, no network address should be required (nor any location service). Note that wireless communication protocols generally allow local broadcast.

A protection against unwanted messages is wished because unwanted messages could be generated by a deficient third party application or by a malicious user on a neighbor node.

4 The Airplug framework

In this section we present the Airplug framework, which satisfies the previously listed requirements. We discuss some of its implementations in the next section.

Main idea. According to the previous section, we assume that applications are implemented using independent processes on top of a POSIX/Linux operating system. A local application is composed of local process(es) while a distributed application is composed of one process per node.

We prone a message oriented framework for communication between local and remote processes. To remain language agnostic, communications are done using standard input and output of each processes. Indeed any language is able to read its standard input (stdin) and to write on its standard output (stdout). Standard error output (stderr) is used for printing information when necessary (information, warning, errors).

In order to manage both internal activity (including GUI events if any) and reading on stdin, it is important to avoid blocking reading of stdin. A common way is to implement asynchronous reading of the standard input. Note that even shell-scripting languages are able to implement this.

An implementation of the framework will have in charge to route the messages from the sending process to the receiving processes, either locally or remotely.

Message format. We prone ASCII text messages for portability. In case binary data have to be sent, they are encoded. Using `yencode` only adds 3% overhead.

A message will contain several fields. To be more flexible, the field size is not specified. For instance, in a UAV squadron, a one-character-wide identifier field is sufficient while it should be larger for vehicles. Fields are then separated by a specific character. As few fields are required, this technique saves space.

The field delimiter character is forbidden inside a field. Then it should be preferable to be able to change it dynamically in case of conflict for some messages. For this purpose each message begins by its field delimiter character. At the reception, the first step consists in reading the first character. Then the fields are retrieved by splitting the message according to this delimiter.

Basically, the fields are the following: sending host, sending application, receiving host, receiving application, action field, control field, payload. However, not all messages require all these fields. For instance, local messages do not need host fields. Moreover in some cases it is possible to drastically reduce the number of fields and implementations could propose this functionality both for efficiency and simplicity purpose. Indeed, when prototyping a new protocol, working only with the payload is very interesting.

Addressing scheme. Applications are denoted by their name which should be locally unique. A distributed application is composed of one instance (of the same name) on each involved node. Hence, an application instance is uniquely determined by its host name and application name.

Messages can be addressed to a given application or to all, using the keyword `ALL`: in case Application `A` sends a message to `ALL`, then only applications which previously subscribed to messages published by `A` will receive it.

Hosts can be designated by any identifier (including addresses or logical names). At the reception of a message sent by a remote application, the framework drops messages that does not contain the local host identifier. However, as explained in the previous section, inter-node communications will mainly rely on broadcast in the neighborhood. Hence, three keywords are provided for the host: `AIR` for broadcasting to all neighbor nodes, `LCH` (standing for localhost) for sending to local application(s) and `ALL` for both local and remote communication.

For protecting the node against spams, an Application `A` should subscribe to the remote Application `B` for receiving its messages, even if they are sent to Application `A` only (remote direct message `B`→`A`). However, an application will receive the messages addressed to itself if the sender is a local application. This principle of relative confidence locally and no confidence remotely is due to the fact that a local application can easily be controlled to the contrary of a remote application, that may be launched by a malicious user.

5 The Airplug Software Distribution

In this section, we describe the Airplug Software Distribution, which provides several implementations of the Airplug framework. These different implementations (called *mode*) are complementary (lab study, road tests...) and compatible: an application can indifferently be used in any of these modes.

5.1 Airplug-term: the terminal mode

As the Airplug framework relies on standard IO for inter-process communication, it is worth noting that UNIX shell offers a (limited) implementation of the Airplug framework. For instance, a communication between processes alpha and beta is simply done by `./alpha | ./beta`. Bidirectional communication can be implemented thanks to *named piped* (command `mkfifo`) and multiple receivers thanks to the `tee` command. Any topology can then be drawn using shell facilities.

Based on this principle, the Airplug-term is an implementation of the Airplug framework to be used in a UNIX terminal. It is well adapted for rapid prototyping and proposes many features through some libraries.

Currently these libraries are dedicated to Tcl/Tk programs but they could easily be extended to other languages. Note that Tcl/Tk proposes high level data structures, many optional libraries and allows rapid GUI design (useful when prototyping). This language is also used by ns-2 [15] and the adaptation of an Airplug application for this network simulator is easy using the Airplug-ns mode, a set of ns-2 add-ons [13]. An Airplug-term option allows to switch from graphical programs using Tk to Tcl programs only, for embedded screen-less computers without graphical libraries.

The Airplug-term mode proposes an implementation of the extensible message format through three messages types. With the `what` type, only the payload is sent. This is used for instance for prototyping a distributed application (e.g., a distributed data collection application). As there is a single application per node, the messages can only contain the payload: each process launched from the terminal represents a node.

With the `whatwho` type, only the payload and the sending and receiving applications are sent. This is useful when designing a distributed application that needs to interact to another one. For instance to design a geographic protocol named GEO which requires geographic positions provided by a local application named GPS, one needs to distinguish between GEO $\leftrightarrow$ GEO communication and GPS $\rightarrow$ GEO communication.

The `whatwhowhere` message type includes the host field (where) and represents the complete message format. It is used when a distributed application relies on the result of another one. For instance, a distributed vocal chat application could use a distributed group membership service.

Note that, when an application designed with a given message type is used in another scenario, Airplug-term proposes an automatic adaption of the messages so that in almost all cases it is not necessary to modify the application.

5.2 Airplug-emu: the emulation mode

Using the shell facilities, any network topology can be drawn, even dynamic ones. Indeed some named pipes can be removed and others can be added according to nodes mobility.

The Airplug-emu mode automatizes this principle [1]. This tool is a *network emulator*: all the upper layers are identical to real experiments while the low layers (wireless communication) are artificially reproduced using shell facilities.

An XML file allows to describe the scenario: applications running on each node, movements of each node, and so one. Then, by analyzing the nodes position, Airplug-emu modifies the connections between processes. It is possible to dynamically modify the range and the reliability of the communication, the loss rate or the delay. Airplug-emu can download maps (Open Street Map) to display the moving nodes along the roads for instance.

Several kind of mobility patterns can be used, including GPS log files obtained during road tests for instance or ns-2 traces. Moves can be accelerated or slowed down dynamically. Obviously Airplug-emu can also be used for non moving nodes (e.g., wireless sensor networks).

This network emulator allows to prepare real experiment as well as to replay them while varying some parameters. This is very convenient to save time. We showed that, when injecting delays and loss rates measures saved during a real test, then very accurate results are reproduced in the lab by Airplug-emu [1].

Airplug-emu can also be used with imaginary scenarios, and can be used to test critical situations or scalability of a protocol. The limit in terms of scenario complexity is given by the capacity of the computer to manage processes. For instance, for a scenario including two convoys of 5 vehicles crossing each other with one alert application and one multi-hop protocol per vehicle [3], about 100 processes are used (including communications). Such a scenario runs easily on a Linux laptop. Note that Linux kernel is generally able to manage up to 32000 processes and can run on powerful servers.

Finally, Airplug-emu can be extended to multiple computers using the Airplug-rmt mode. This mode allows remote execution of some applications: they are connected by sockets to a specific application named RMT that relay their messages between computers. Besides extending the capacity of the emulator, it also allows to include real wireless connection inside the emulation, leading to an *hybrid emulation.*

5.3 Airplug-live: the experiment mode

For real use, an efficient implementation of the Airplug framework is proposed, named Airplug-live. It is composed of a core program in C managing both local and inter-nodes communications [5]. This program allows to reuse the applications prototyped using Airplug-term and studied using Airplug-emu without any modification. It acts as a middleware between the applications and the network interfaces, while still running on user mode on top of a Linux OS.

The Airplug core program launches itself the local applications running as independent processes. For each of them, standard IO are redirected from and to the core program. Then, each time a local application writes on its standard output, the Airplug core program receives the data, and reciprocally. This program is also in charge of the remote communication using networking facilities available on the node.

The Airplug core program scrutinizes the links from the local applications as well as the network interfaces and forwards the message to the appropriate destination, following the Airplug addressing scheme. A careful attention has been paid on its robustness. The program relies only on the standard `libc` library and is compiled with `gcc`. The executable is less than 40 KB. The source code has less than 3300 lines for 177 KB. This implementation of the Airplug framework is light, robust and portable.

6 Validation

Validation concerns both the implementation and the overall framework proposal.

A complete framework. After analyzing the consequences of the dynamic networks on the distributed applications (Section 2), we drawn some implementation rules in Section 3. We then proposed the Airplug framework in Section 4.

In the previous section, three implementations of the Airplug framework have been introduced: the prototyping mode Airplug-term relying on the shell facilities, the studying mode Airplug-emu providing a dynamic network emulation and the real use mode Airplug-live running on communicating terminals or PC. This indicates that a process based framework running on top of operating systems and relying on light messages conventions is convenient for reusing existing tools (e.g., shell) and to be compliant to any programming language and paradigm.

An application designing for one of this mode can be used for all the others without modification. Additionally, Airplug-ns is a mode composed with add-on for Network Simulator [15] and Airplug-rmt is a mode allowing remote execution. Hence the Airplug Software Distribution offers a complete set of tools for designing applications for dynamic networks, including rapid prototyping, in-lab tests, real tests, replay of the test and scalability studies by emulation.

Airplug-live has been compiled for several architectures, including ARM. It has been used in vehicles, road side unit (RSU) or unmanned aerial vehicle (UAV). This shows that the Airplug framework is light and portable.

A powerful framework. The framework could be complete and portable while not being really useful and this point is the most important. We show the usefulness of the framework through applications.

About fifty prototypes have been designed for Airplug. Some applications are dedicated to local devices such as GPS, Bluetooth or ZigBee. Some of them are user centric applications such as distributed games or chat. They rely on distributed services.

As the Airplug framework has been used for studying dynamic networks, dedicated applications have been designed, mainly for vehicular networks (an emblematic case of dynamic networks): neighborhood analyzing, reliable diffusion, multi-hop communications, gateway discovering, unicast communication, performance analyzing [12]... In particular, we implemented a powerful one-to-many multi-hop communication protocol relying on conditions instead of addresses [8]. It can route alert messages according to trajectory correlation, GPS position, date and many other logical conditions. We also proposed an opportunistic vehicle to infrastructure architecture, exploiting any road side unit, close vehicles equipped with 3G or public WiFi hot-spot [10]. We also implemented a unicast private communication between phones in distant vehicles relying only on neighbor-to-neighbor communications [11].

Some complex applications provide distributed services besides the dynamic of the network: dynamic group membership [9], distributed data collect [2], distributed data fusion [6]... All these algorithms are self-stabilizing meaning that they can support transient failures. Some of them still guarantee a continuity of services between failures.

Most of these applications have been tested on the road and studied by emulation. The reader can be seen some animated screenshot captures as well road-tests movies in the Airplug Software Distribution web site [4]. These examples show that, while simple, the framework allows to design robust complex and varied applications, without any requirements on the programming language.

7 Related work

Many message passing frameworks already exist, such as ONC RPC, CORBA, Java RMI, DCOM, SOAP, .NET Remoting... The main interest is to specify nothing internally: only the interfaces have to be specified. However, such frameworks are not dedicated to dynamic networks and they implement many services not pertinent in this specific context. To the contrary the Airplug framework is very light and portable, does not rely on TCP/IP and can work even when the network is highly dynamic.

Popular frameworks generally limits the languages and the programming paradigm. For instance, the OSGi framework relies on modules, a more advanced concept than processes, but is dedicated to Java and requires then a JVM installed. To the contrary, the Airplug framework remains language agnostic.

The subscription mechanism in Airplug is used for one-to-many diffusion and also for protection against spams that may generated by malicious users or deficient equipments. Similar mechanisms can be found in JMS for instance.

The Airplug message format is based on fields separated by a delimiter character announced at the beginning of the message. This allows adaption to the number of hosts while saving space. By comparison, IPv4 uses 32 bits for addresses, which appeared to be too short for Internet and too much for small and close testbeds.

The application naming convention in the Airplug framework is close to the Provider Service Identifier (PSID) used in the IEEE WAVE architecture [14].

The opportunistic communication paradigm is proposed in WAVE through the WAVE Short Message Protocol. A node can announce some services and also broadcasts some messages to close neighbors that may be interested [14]. However the Airplug framework should not be compared with WAVE. Instead, it should be seen as a distributed framework able to efficiently run on top of the WAVE stack.

Finally the main drawback of Airplug is certainly to not be compatible with other existing frameworks nor to provide as many services as them. However, this can be solved by implementing stubs for compatibility. The fact that any application reading on stdin and writing on stdout can be used with Airplug allows many complementary usages. Moreover, new services could also be added in Airplug, providing their are realistic in dynamic networks. A common way for this would be to design a new application offering the wishing service.

8 Conclusion

In dynamic networks, a communication link can be used for sending very few successive messages before disappearing. These strong conditions are challenging for designing distributed applications and protocols. They appear when the nodes mobility is high or the capacity of the communication protocol is low (or both), which can happen in vehicular networks, robot networks and more generally in any scenario with communicating and moving terminals.

Many works have been done in the protocols, less in distributed algorithms and very few regarding implementation. Popular frameworks are generally based on TCP/IP and admit limitations in dynamic networks. Still, they often impose constraints that may limit their deployment on target (embedded) computers.

In this paper, we analyzed the requirements for programming distributed applications in dynamic networks. We then proposed the Airplug framework. This light message passing framework relies on simple conventions regarding the message format and the specific addressing. We showed that it can easily be implemented on user-space on top of standard POSIX operating systems. It is very portable and accepts any programming paradigm and any language.

The Airplug Software Distribution is a convenient set of tools (and applications) for designing and deploying complex scenarios in dynamic networks from scratch. It includes several implementations of the Airplug framework, for rapid prototyping, real tests or in-lab studies using network emulation. An Airplug application can run indifferently on all these implementations.

Many developments and experiments validate our framework specification as well as the efficiency of its implementations, for communication, data management, nodes control, end-user entertainment... Airplug is also used for teaching distributed computing and dynamic networks.

Future work will concern stubs development for compatibility with other frameworks and experiments with IEEE WSMP and other ITS protocols.

Acknowledgment. Some of the mentioned applications have been studied with my students since 2004. Road testbeds have been done with the help of the Heudiasyc engineers and colleagues. Many thanks to all of them.

References

1. A. Buisset, B. Ducourthial, F. El Ali, and S. Khalfallah. Vehicular networks emulation. In IEEE ICCCN'10, Zurich, Switzerland, August 2010.
2. Y. Dieudonné, B. Ducourthial, and S.-M. Senouci. COL: A data collection protocol for VANET. In *IEEE Intelligent Vehicles Symposium (IV 2012)*, June 2012.
3. https://www.hds.utc.fr/airplug/doku.php?id=en:doc:movies:start.
4. https://www.hds.utc.fr/airplug.
5. B. Ducourthial. About efficiency in wireless communication frameworks on vehicular networks (invited paper). In ACM WIN-ITS workshop, Vancouver, 2007.
6. B. Ducourthial, V. Cherfaoui, and T. Denoeux. Self-stabilizing distributed datafusion. In SSS'12, Toronto, October 2012.
7. B. Ducourthial and F. El Ali. Characterizing dynamic networks. Lab. Heudiasyc, Université de Technologie de Compiègne, France, 2013. submitted.
8. B. Ducourthial, Y. Khaled, and M. Shawky. Conditional transmissions: a communication strategy for highly dynamic vehicular ad hoc networks. *IEEE TVT*, 56(6):3348–3357, November 2007.
9. B. Ducourthial, S. Khalfallah, and F. Petit. Best-effort group service in dynamic networks. In *22nd ACM SPAA'10*, Greece, June 2010.
10. F. El Ali and B. Ducourthial. A light architecture for opportunistic vehicle-to-infrastructure communications. In *8th ACM International Symposium on Mobility Management and Wireless Access (Mobiwac 2010)*, Bodrum, Turkey, 2010.
11. F. El Ali and B. Ducourthial. A distributed algorithm for path maintaining in dynamic networks. In *International Workshop on Dynamicity (DYNAM'11), collocated with (OPODIS'11)*, Toulouse, France, December 2011.
12. F. El Ali, B. Ducourthial, and S.-M. Senouci. On the capacity of communications in a convoy of vehicles. In *73rd IEEE VTC-Spring*, Budapest, Hungary, May 2011.
13. S. Khalfallah and B. Ducourthial. Bridging the Gap between Simulation and Experimentation in Vehicular Networks. In *72nd IEEE VTC-Fall*, Ottawa, Canada, September 2O1O.
14. Vehicular Environments (WAVE) Working Group of the Intelligent Transport Systems (ITS) Committee. IEEE P1609.3/D7.0: Draft standard for wireless access in vehicular environments (wave) - networking services, June 2010.
15. Network simulator. http://www.isi.edu/nsnam/ns.
16. H. Saleet, R. Langar, O. Basir, and R. Boutaba. Proposal and analysis of region-based location service management protocol for VANETs. In *IEEE Globecom'08*, 2008.