MANET auto-configuration using the 802.11 SSID field in Android

Ozaine Oscar[1], Díaz-Ramírez Arnoldo[1], and Calafate Carlos T.[2]

[1] *Instituto Tecnológico de Mexicali, Av. Tecnológico sn Col. Elías Calles, Mexicali, B.C., México, 21376,*
oscar.ozaine@gmail.com , adiaz@itmexicali.edu.mx
[2] *Universitat Politècnica de València, Camino de Vera sn, Valencia, España, 46022*
calafate@disca.upv.es

Abstract: Mobile *Ad hoc* networks (MANETs) enable the communication between mobile nodes via multi-hop wireless routes. The main objective is to extend the connectivity range of nodes through packet forwarding, thereby avoiding the use of a fixed infrastructure.

In this paper, we introduce a previously proposed solution for the auto-configuration of IEEE 802.11 based MANETs that relies on SSID parameter embedding. This solution allows users to join an existing MANET without resorting on any additional technology, and even in the presence of encrypted communications. The proposed method was implemented in the Android operating system since it is widely used in mobile devices. Implementation details are discussed, and a performance analysis is included, comparing the proposed Android implementation against a standard implementation in a personal computer running the Linux operating system.

Key Words: Wireless Network, MANET, Wi-Fi, Network Configuration, Android

1. INTRODUCTION

A MANET, also known as a Mobile *Ad hoc* Network, is a collection of wireless nodes that can dynamically organize themselves in an arbitrary and temporal topology, creating a network without the need of using a predefined infrastructure (Chlamtac et al., 2003). Fig. 1 (up) shows an infrastructure network (e.g. Wi-Fi) that depends on the existence of a central node or access point. In contrast, a MANET allows the communication between mobile devices without the need of an access point, as seen in Fig 1 (bottom). The main advantages of MANETs are their cost and flexibility. However, despite of these advantages, their deployment is not easy. For instance, the initial configuration of nodes may be a complex issue.

A completely functional MANET needs all the stations to be configured using coherent network parameters at the different protocol layers. The 802.11 standard defines the MAC (media access control) and physical layer (PHY), the latter including self-configuring parameters such as the type of modulation, the frequency, and time synchronization.

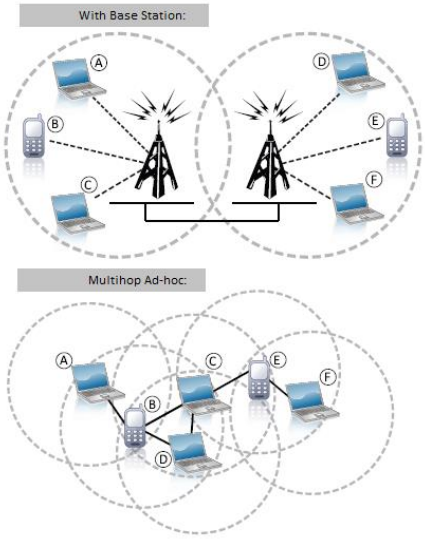

Fig. 1. Infrastructure network and a MANET

The most critical parameters defined at the MAC layer are: service set identifier (SSID), energy saving mode, encryption type, and encryption key. In the

network layer, the most relevant parameters are: version of the IP protocol used (IPv4 or IPv6), IP address of the node, network mask, routing protocol, and gateway. Notice that some of these parameters must be set by the user. Since there is no central node or access point to provide configuration support, the complexity involved in configuring a MANET can become a limiting factor for most users.

Recently, a solution was proposed to solve the MANET auto-configuration problem using the 802.11 SSID field (Villanueva et al., 2013). The aim of the proposed solution is to offer a configuration on the layer-2 and layer-3 of a complete decentralized MANET, which does not require any other technology in addition to the IEEE 802.11. This paper introduces the auto-configuration proposal and discusses its implementation in the Android operating system, which is widely used in mobile devices. Also, a comparative analysis of our implementation and an implementation in a computer running the Linux operating system is provided.

The rest of the paper is organized as follows. The auto-configuration method for MANETs is introduced in Section 2. Details of our implementation for the Android operating system are discussed in Section 3. Section 4 shows the evaluation results of our implementation compared to a standard Linux-based implementation. Finally, Section 5 concludes the paper and discusses future works.

2. AUTO-CONFIGURATION METHOD FOR MANETS

The successful deployment of a MANET depends mostly on the physical and MAC layers of the 802.11 standard, as well as on network-level configuration. The associated complexity is further increased since MANETs do not have any kind of centralized management for nodes attempting to join the network. The characteristics of MANETs (variable topology, decentralized, short duration) complicate the process of configuration since the parameters of the participating nodes can change frequently. Additionally, they need support for routing protocols in all the nodes due to the MANET's requirement for multi-hop communication.

In this paper we focus on the auto-configuration method for MANETs proposed by the Computer Network Group from the Polytechnic University of Valencia, Spain (Villanueva et al., 2013), which offers a fully decentralized MANET configuration that does not require any additional technology besides the IEEE 802.11, and does not introduce any extra traffic overhead. It is a novel solution since it addresses both layer-2 and layer-3 configuration requirements, while avoiding the limitations of similar proposals published in the literature.

The proposed solution considers the 802.11 standard since it is broadly used in MANETs. With this technology, although communication may be encrypted, the beacon frames are not. Thus, the proposal is based on the use of the beacon frames to configure the required parameters for nodes attempting to join the MANET.

Inside the beacon frames there is just one field that is not set automatically: the SSID field. The user defines the value of this field, setting the name of the network using a maximum of 32 bytes, according to the 802.11 standard.

In the proposed solution, the SSID is modified and divided into four blocks, including not just the name of the network, but also the connection parameters that allow the configuration to be completed in a transparent way. Thus, the SSID is divided as follows:

- Block 1: Name of the network.
- Block 2: Properties of the network.
- Block 3: Sub-network prefix.
- Block 4: Seed of the session key.

2.1 Proposed SSID partition

The first block refers to the name of the network, and its length is from 3 to 12 bytes long. In order to distinguish regular beacons from the formatted beacons, a special character (byte) has been inserted at the beginning of the SSID field, to identify when a MANET is using the auto-configuration method. This special non-printable character can have only two values, one of them indicating that the IP version used is IPv4, whereas the other one indicates that the MANET is using IPv6. Next to this special character, the network's name follows, as defined originally in the SSID field. Finally, at the end of the first block, it a non-printable special character appears again.

The second block refers to the properties of the network, and it is conformed by just one byte. This byte is used to define two parameters. The security mode (WEP, WPA-PSK, WPA2-PSK) is defined using the first three bits. The routing protocol is defined using four additional bits. Some of the options are: OLSR (Clausen and Jacquet. 2003), AODV (Perkins et al. 2003), DYMO (Chakeres and Perkins. 2008), and DSR.

ISBN: 978-607-95534-5-6

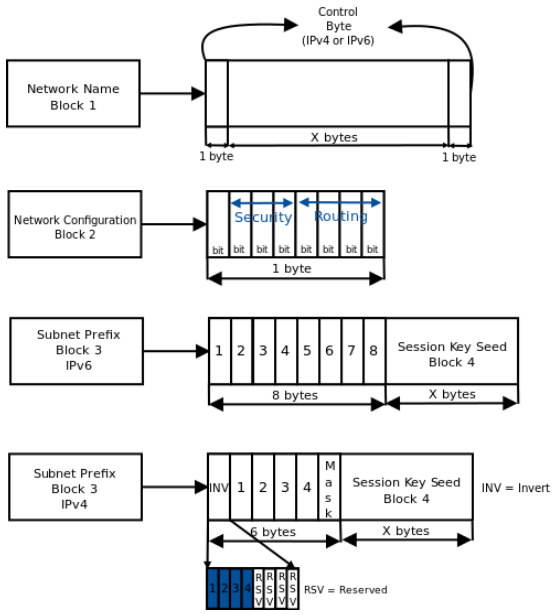

Fig. 2. Detailed SSID partitioning strategy

The third block contains the IP address. If IPv6 is used, the IP address is defined using eight bytes, that represent the first half in the Unique Local Unicast range of addresses (FC00::/7). The latter eight bytes are derived from the MAC address of the wireless network interface according to the strategy defined in RFC 4291.

If IPv4 is used, the node's address is identified with six bytes, where the first byte (STD/INV) is used to invert possible null values in any of the four bytes that define the IP address, converting any value 0x00 into 0xFF (0 to 255). Thus, stations attempting to configure themselves must reverse the inverted bytes to recover the original values. The network mask is defined in the last byte.

The fourth block has the seed for the session key. If the network has a security mode, this block is in charge of deriving the real session key to perform the encryption. Fig. 2 illustrates the proposed SSID partition strategy.

2.2 Deriving the session key

The seed size is variable, and depends on the IP version and the network name. Once the network identifier and the IP version are defined, the seed of the session key fills the remaining bytes of the SSID field. Theoretically, the SSID field is composed of 32 bytes. However, the operating systems handle the last byte as a NULL character, reducing it to 31 bytes.

One of the limitations of the seed embedded into the SSID has to do with the handling of NULL values. This means that the number of possible combinations will be slightly reduced by this restriction. Thus, the original space of $256^{(20-x)}$ combinations ($256^{(22-x)}$ for IPv4) is reduced to $255^{(20-x)}$ ($255^{(22-x)}$ for IPv4). In the worst case condition (if the network name uses all 10 characters), there are still about 10^{24} possible seeds (approx. 7×10^{28} for IPv4). This means that the chances that the same seed repeats for a same group of users become negligible.

When the seed of the session key is obtained, the real session key is calculated by a hashing mechanism called MD5. The MD5 mechanism (Rivest, 1992) receives the seed of the session key (random values), and the session key that every MANET user shares.

Since the number of bits in the hash value may be shorter than the one required by the selected security mode, the hash is a recursive function. This means that the output is fed back to generate a new hash value until the key generator gathers enough bits. Depending on the selected security mode, the key generator may have to chop part of the input or perform hashing again, in order to obtain the correct number of bits for encryption. For example, if WEP-40 is used for encrypting, a single hashing round suffices since the 128 bit output is enough to obtain the 40 bits required for a valid session key. In contrast, if WPA-PSK is used, two MD5 hashing rounds are needed to generate a session key of 256 bits.

3. IMPLEMENTATION DETAILS

Mobile devices running Android OS are usually purchased together with subscriptions that include data transfer deals, allowing mobile Internet access. Sometimes, mobile users access to the Internet through other mobile devices thanks to the use of *tethering* applications. These applications provide the functionality to use mobile phones similarly to conventional land-line modems. Phone and computer can be connected by a data cable through USB, or wirelessly through WiFi or Bluetooth

To be able to implement the proposed solution it is necessary to enable the *Ad hoc* mode in the device. This can be done with a *tethering* application called *android-wifi-tether*. It switches the wireless interface to *Ad hoc* mode and provides a user-interface to configure the network parameters, such as the SSID, the static IP address and sub-network. Also, to be able to develop the proposed solution, Android gives the option to develop applications using the Eclipse

ISBN: 978-607-95534-5-6

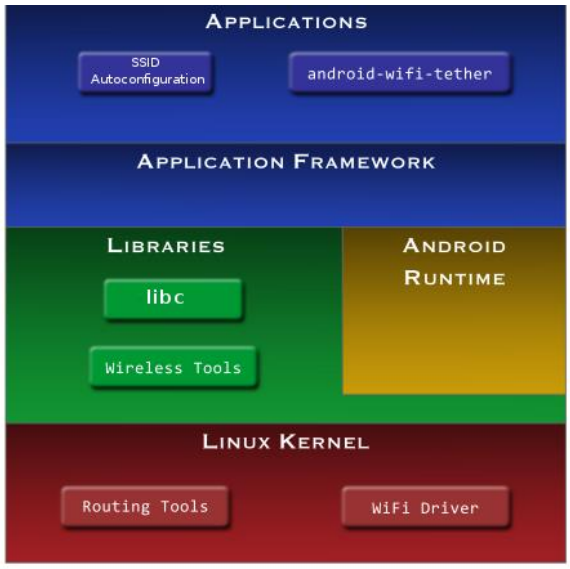

Fig. 3: Auto-configuration components in the Android OS

IDE, using a simple plug-in called ADT (*Android Development Tools*).

With the *Ad hoc* mode enabled in the Android device, there is the need to manage the wireless card. The Wireless Extension API and wireless *tools* by Jean Tourrilhes (Tourrilhes, 2000) enable the user to manage and configure the network parameters. Also, they provide support for *Ad hoc* networks. Fig. 3 shows how the auto-configuration method and the *tethering* application fit in the Android stack.

3.1 Cross compiling the wireless tools

Cross compiling refers to a compiler that runs on one computer but produces object code for a different type of computer. Cross compilers are used to generate software that can run on computers with new architectures or on special-purpose devices that cannot host their own compilers. This technique is used to cross compile the *wireless tools* for Android OS.

The wireless tools are a set of tools allowing manipulating the Wireless Extensions. They use a textual interface, aiming to support the full Wireless Extension. There are two commands that our implementation use:

- *iwconfig*: Manipulates the basic wireless parameters.
- *iwlist*: Allows to initiate scanning and list frequencies, bit-rates, encryption keys, etc.

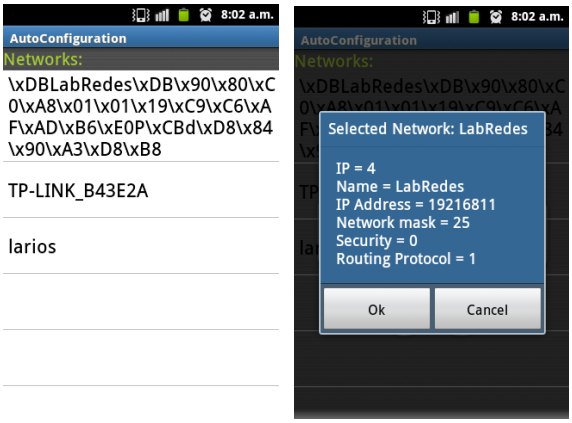

Fig. 4: Network scan screen and scan results

3.2 Obtaining the SSID

The method used to obtain the SSID of an *Ad hoc* network is based entirely on the *iwlist* command. This command is executed on the device, checking all possible networks. The output of this command is a string containing all the information about the network.

The information saved in the string is the physical address of the network, the frequency at which it is operating, the signal level, the SSID, and the encryption method, among others, noteworthy that this parameters are in hexadecimal format, except for the name of the network. The string then is filtered to obtain the SSID field. Having the string filtered and showing the SSIDs of all nearby networks, the implementation identifies the network and the user selects one to get all the connection parameters and eventually connect. Fig. 4 shows the user interface to manage the implementation in the Android OS.

4. EXPERIMENTAL RESULTS

The previous section described the implementation details of the proposed auto-configuration method for the Android OS, which includes two components: one that allows creating the *Ad hoc* network, executed only by the first station, and another one that allows joining an existing auto-configurable MANET. In this section, the performance results obtained when comparing the Android device implementation and the Linux PC implementation are discussed.

Performance measurements were made using a mobile device running the Android OS, v2.3 (Gingerbread) with an 800 MHz ARM 11 processor, and 278 MB of RAM. The application running on this device was developed using the Java

programming language. On the other hand, a middle-range netbook computer equipped with a single Atom Intel 1.6 GHz processor, with 1GB of RAM, and running the Ubuntu 12.04 OS was also used. The application running on this netbook computer was developed using the *C* programming language. Both wireless cards supported the IEEE 802.11 standard, and both were within the transmission range of the terminal that initially creates the *Ad hoc* network.

Table 1 shows the average time required by both implementations when performing several tasks related to the auto-configuration of the node. Concerning the time required to obtain the configuration parameters, the applications merely parse the beacon received to extract configuration information. This task is achieved in just 4 μs for the Ubuntu implementation, but for the Android implementation it took 5,720 μs. The significant difference detected when performing this task is associated to the method used to retrieve the configuration. In Ubuntu it was developed using the *C* programming language, and uses the 802.11 communication socket directly. On the other hand, the implementation in the Android device was developed using the Java programming language, and it obtains the configuration with the *iwlist* command. Thus, it needs to filter the output of this command and obtain the SSID, which introduces some overhead. The reason for adopting this method is that the Android OS, for security reasons, blocks all the 802.11 communication sockets. It is shown that, for the rest of the tasks, the Ubuntu implementation performed approximately twice as fast as the Android implementation. This is due to the difference in the processor performance of the devices.

After performing the experiments and analyzing the obtained results, we can confirm that the proposed solution operates correctly since the nodes were properly configured. Also, it is scalable by design since the beacon generation process is distributed, meaning that all participant nodes may generate beacons at any instant of time. Thus, new nodes approaching the MANET can become part of the *Ad hoc* network as soon as they receive a beacon frame. The proposed solution becomes highly effective even in large-sized and highly disperse MANET environments.

5 CONCLUSIONS AND FUTURE WORK

Mobile *Ad hoc* networks do not rely on a centralized node to be created and operate. However, their decentralized structure complicates the initial configuration of their nodes. Recently, an auto-configuration method for MANETs was proposed.

Table 1: Comparative between the Android and GNU/Linux implementations

	GNU/Linux	Android
Obtain Configuration	4 μs	5720 μs
Establish SSID without security	4400 μs	8880 μs
Establish SSID with security(WEP)	4600 μs	12920 μs
Establish IP Address	4000 μs	8055 μs

The solution is based on the SSID field that is present in beacon frames. They are periodically generated for the 802.11 station to promote basic information of the network. By listening and interpreting these beacons, new stations will be able to determine all the connection parameters to join the MANET.

In this paper, the implementation details of the auto-configuration method using the Android OS are provided, and its performance is compared against an implementation in the Linux OS for reference.

In order to implement it in Android OS, several tools were needed since Android does not include them, such as the *Ad hoc* mode and the wireless card management tools. Also, a prototype in Java was developed and used in a mobile device running the Android OS. The Linux implementation outperformed the Android implementation, although the latter is within perfectly acceptable performance bounds. Also, the auto-configuration method performed correctly in both architectures. Additionally, we conclude that the method is scalable by design since the beacon generation process is distributed, meaning that all participant nodes may generate beacons at any instant of time.

As future work, we plan to improve the Android implementation and evaluate the solution using other mobile devices under difference scenario settings.

REFERENCES

Android SDK official web site:http://developer.android.com/sdk/

David B. Johnson, *et al*: *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)*, Internet Draft, 2007. URL http://tools.ietf.org/html/rfc4728

I. Chakeres, C. Perkins: "*Dynamic MANET On-demand (DYMO) Routing*", Internet Draft, 2008.

Imrich Chlamtac *et al.*: "*Mobile ad hoc networking: imperatives and challenges*", Ad Hoc Networks, pp. 13—64, 2003.

J. Tourrilhes: Wireless Tools for Linux, 2000.

Ma José Villanueva *et al.*: "*Seamless MANET autoconfiguration through enhanced 802.11*

beaconing", Mobile Information systems Journal. Volume 9, Issue 1, 2013..

Samir R Das *et al.*: "*Ad hoc on-demand distance vector (AODV) routing*", Internet Draft, 2003, URL http://www.ietf.org/rfc/rfc3561.txt

Philippe Jacquet: "*Optimized link state routing protocol (OLSR)*", Internet Draft, 2003, URL http://www.ietf.org/rfc/rfc3626.txt

Ronald L Rivest, others: RFC 1321: *The MD5 message-digest algorithm*. April, 1992.