An Ad-hoc Network of Android Phones Using B.A.T.M.A.N.

Project Report SPVC-E2010, IT-University Copenhagen

Leo Sicard
lnsi@itu.dk

Matyas Markovics
mama@itu.dk

Giannakis Manthios
gima@itu.dk

ABSTRACT

Mobile telephony has become a common thing, and is generally taken for granted in developed countries. However, there are situations in which it is unavailable, because of the lack, or failure, of the necessary infrastructure. In developing societies, it is an expensive service to most. In this paper we demonstrate that telephony can be made independent of mobile operators and costly infrastructure. Mobile phones have evolved to become small devices with strong computational power and enhanced networking capabilities. By adapting cutting-edge networking software to these devices, we can make telephony autonomous and free. We describe our experiences with applying the B.A.T.M.A.N. routing protocol to smartphones running Android OS. We evaluate the performance and reliability of the self-established mesh network, and discuss the remaining steps to free ad-hoc telephony.

INTRODUCTION

Mobile telephony has become available to all, thanks to its success as a technology and as a commercial enterprise. Devices have gradually become smaller, and at the same time more powerful, offering a multitude of features to the user, and a whole set of networking options. Today, the least we expect from our phone is the capability to ring someone, regardless of time and location. However, this is not always possible. There are situations in which mobile telephony is unavailable, either because of the lack of necessary infrastructure, e.g. underground or in sparsely populated areas, or because of the failure of same infrastructure, e.g. in disaster situations. Furthermore, there are contexts in which mobile telephony is expensive. In most developed countries, its cost is considered trivial, but in poorer communities, mobile telephony can be the source of concrete economical limitations. The basic feature of 'making a mobile phone call for free, at anytime and anywhere' is the focus of this paper. We can reach this goal by porting mesh networking to Wi-Fi enabled mobile phones.

Wireless ad-hoc networks are formed by autonomous interconnected nodes. They have the key features of being self-organizing and self-healing, removing the need for a centralized infrastructure. They are often static, as in the deployment of sensor networks for the monitoring of physical or environmental conditions. With the advances in wireless technologies and the miniaturization of mobile computing devices, Mobile Ad-hoc Networks have emerged (MANET), introducing the extra challenge of mobility to those networks. Recent research in routing protocol design addresses issues that are common to mesh networks, i.e. mobility and resource limitations.

By implementing the B.A.T.M.A.N. routing protocol [22] (Better Approach To Mobile Ad-hoc Networking) on open source wireless routers connected to Analog Telephony Adapters (ATA), it is possible to deploy community networks that allow for cheap access to traditional telephony, as shown by the Village Telco initiative [9]. By porting B.A.T.M.A.N. to Android OS, mobile phones can connect to such networks, thus providing the user with free mobile telephony. This is what the researchers at the Serval Project [13] have achieved this year.

Our focus is the implementation of a network that addresses situations in which the infrastructure is absent, and the port of B.A.T.M.A.N. to the 2.1 version of the Android OS. We are also providing documentation and evaluation results on the behavior of the routing protocol. The main challenge lies in accessing the Linux subsystem of the mobile devices, in order to enable wireless ad-hoc networking under the control of the B.A.T.M.A.N. daemon. On an Android 2.1 HTC Legend phone, this implies removing all access restrictions, enabling wireless ad-hoc mode, and integrating the B.A.T.M.A.N. daemon to the Android OS using an Android developed application. Following this, we have performed experimental tests on the behavior of the interconnected mobile devices, including single hop and multiple hop routing, join and leave operations and dynamic reconfiguration of the network.

With this paper, we contribute to the evolving field of mobile ad-hoc networking, and more precisely the efforts that are made to enable free mobile telephony. By implementing, documenting and evaluating this system, we help determining the extents and limitations of ad-hoc networking with Android phones. We start by presenting the Related Work. Then we elaborate on the Background and Research Methods of the project. We describe the System Design and present our experimental results in the Evaluation. This is

Paper submitted for evaluation in the Pervasive Computing Course, Fall 2010. The IT University of Copenhagen.

followed by a short Discussion and finally we sum up our study in the Conclusion.

RELATED WORK

Wireless ad-hoc networks have distinct characteristics. They consist of interconnected nodes that act as autonomous routers, allowing them to be self-forming and self-healing. They do not rely on any centralized infrastructure, and are flexible in their topology. Mobile Ad-hoc Networks (MANET) form a subcategory of ad-hoc networks. They are subject to dynamic topology changes, because the participating nodes are mobile, adding new complexity to the system. Wireless Mesh Networks (WMN) usually refer to networks that are composed of static routers that perform the routing functionality for client devices which connect to them. A broadband community wireless network is a good example of a wireless mesh network.

There is an ongoing effort to design specifically fitted routing protocols that allow for dynamic topology changes without being too resource-intensive. The most frequently encountered routing protocols for ad-hoc networks belong to the following two categories: proactive (table-driven) and reactive (on demand). The Optimized Link State Routing Protocol (OLSR) [18] is a proactive protocol. It maintains network topology knowledge by periodically flooding the network with updated routing tables. The Dynamic Source Routing (DSR) [20] and Ad-hoc On-demand Distance Vector (AODV) [23] protocols are examples of reactive protocols, where routes are found on demand by flooding the network with route request packets. The main disadvantage of such protocols is the overhead in bandwidth and power caused by the amount and frequency of exchanged data, as well as heavy computations at each node. It has also been shown that OLSR lacks in performance when networks grow too large. B.A.T.M.A.N. [22] was created to address those shortcomings, as a routing algorithm for a large static mesh. The intelligence of B.A.T.M.A.N. lies in the fact that the nodes do not have global knowledge of the rest of the network, but only "best neighbor" knowledge, i.e. what the best next hop option is, depending of the node that you want to reach.

The emergence of low-cost open source wireless Access Points (AP) such as the Linksys WRT54G series [11], have fueled a series of initiatives to implement wireless community networks, in order to offer an alternative to traditionally proprietary telephony and internet services. A good example of this type of venture is Village Telco [15]. They have developed the Mesh Potato [25], which is a simple wireless AP connected to an Analog Telephony Adapter (ATA) and running a Linux based firwmware called OpenWRT [12], as well as the B.A.T.M.A.N. mesh networking software. This lightweight infrastructure allows them to deploy mesh networks to bring cheap telephony and internet to whole communities.

The people at the Serval Project have developed the Batphone, that builds upon the Village Telco technology. It is an Android smartphone running the B.A.T.M.A.N. daemon, as well as Asterisk [3], the Session Initiation Protocol (SIP) [24], and DNA (Distributed Numbering Architecture) [26]. The device can integrate to an already (or easily) deployed network of Mesh Potatoes, and provide free ad-hoc telephony services to otherwise traditional mobile phones. This last aspect is very important in relation to disaster situations where all other communication means are missing.

The swedish company Terranet [14] is an example of the growing interest for ad-hoc telephony, and more generally wireless peer-to-peer technologies.

BACKGROUND AND RESEARCH METHODS

We are a group of master students at the IT-University of Copenhagen. We have worked during the fall 2010 semester with the support of the Pervasive Interaction Technology Lab (PITLab) and the supervision of our teachers.

In this section, we begin with a presentation of our experimental setup. We then continue with the description of the test scenarios that have been performed. Finally, we mention the data collection process.

Technical and Experimental Setup

Our purpose is to implement and run the B.A.T.M.A.N. routing protocol on mobile devices, in the absence of any infrastructure. Therefore, the minimum testbed consists of mobile devices only. In this case, we have used 2 HTC Legend phones running Android OS 2.1 that were provided by the PITLab. Moreover, we have used an Apple MacBook Pro computer running Linux Gentoo as a third participating node, and in order to install and run software on the phones, test connectivity, as well as for data gathering. All the phases of this project have been conducted indoor, in the building of the IT-University of Copenhagen.

Test Scenarios

In order to be able to give a precise description of the behavior of B.A.T.M.A.N., we have designed a series of four test scenarios, as described in Figure 1. The scenarios were run on the two Android devices and the laptop computer. Data was collected in the form of routing debug logs generated by the B.A.T.M.A.N. routing daemon.

When discussing routing logic, we use the term *best next hop* to determine the routing knowledge of a node N. For instance, if a node N knows that the best next hop to reach node P is node O, it will send to node O any packet that has node P as destination. We will explain the B.A.T.M.A.N routing logic in further details when presenting the system design.

(1) The Single Hop Scenario
In this scenario, we aim at verifying that single hop routing is correctly performed by the B.A.T.M.A.N. daemon, in a static configuration. We do this by forming a fully connected mesh network between the mobile devices A, B and C. We then check that all nodes have knowledge of each other, and that their routing information is correct.

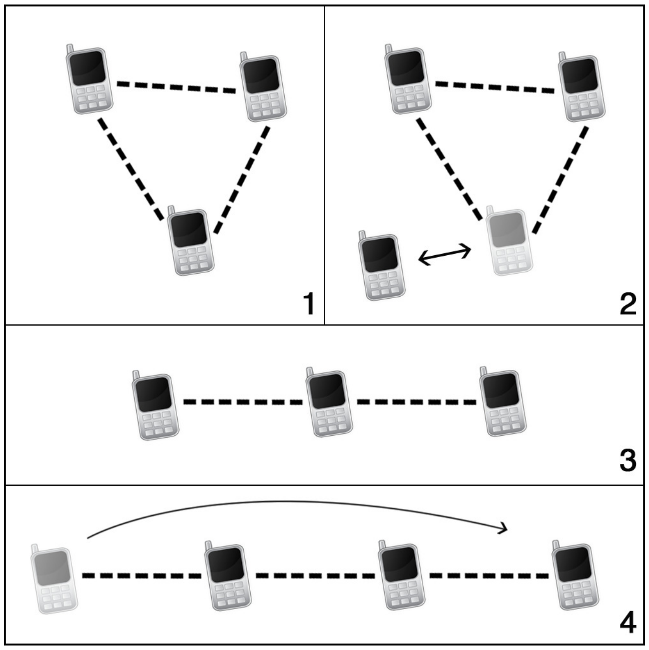

Figure 1. An overview of the test scenarios

(2) The Join / Leave Scenario
Here, we use the existing mesh network from Scenario 1, but introduce network topology changes. One of the devices sequentially leaves and joins the mesh. We verify this behavior by checking that the dynamic node actually disappears and reappears from the routing logs of the other nodes.

(3) The Multi Hop Scenario
The purpose of this scenario is to verify that multi hop routing is correctly performed by the B.A.T.M.A.N. daemon, in a static configuration. In order to do this, we place the devices in a line configuration, so that device A will have to use device B in order to communicate with device C.

(4) The Network Reconfiguration Scenario
In this last test scenario, we reuse the configuration of Scenario 3, but introduce mobility and network reconfiguration. We do this by physically moving the devices so that device C becomes the "bridging" device instead of device B, thus preserving network connectivity.

Data gathering

In order to evaluate the logic and performance of the B.A.T.-M.A.N. routing protocol, we collect the routing debug logs that are generated by the daemon on each device. To do this, we use the ability to connect to the devices as clients with accessory laptop computers, and accessing live debug output through the Android Debug Bridge (ADB) shell.

SYSTEM DESIGN

The system in this case is the network of independent nodes that communicate. The communication is done through UDP packets addressed to broadcast IP of the network. The nodes adjust their behaviour due to the received information. A node consists of 3 components: (1) an operating system specifically for mobile devices, (2) a wireless interface running in ad-hoc / unmanaged mode, (3) a service that implements a routing protocol for mesh networks. It would make sense to have a user application that makes use of the established ad-hoc network as a fourth component. However, the development of such an application was considered out of the scope of this project.

The nodes used are mobile phones that run Android 2.1 with modified access rights. Figure 2 gives an overview of our system and how it fits in the Android OS. Originally, Android does not support the ad-hoc mode of the Wireless interface. Therefore, the application android-wifi-tether [21] was installed. In order to install and run this application, it was necessary to gain root access on the mobile device. The application re-initializes the wireless interface with a predefined ESSID and IP address. Once that is done, other devices can connect to it and establish peer-to-peer communications. However, this setup has an important limitation, which is that it only allows for the pairing of 2 devices. This limitation is addressed by mesh Mobile Ad-hoc NETworks (MANET). Nodes in a MANET can reconfigure themselves in order to communicate with more than one device. They also maintain routing rules and a dataset to be able to reach devices in their neighbourhood. The B.A.T.M.A.N. protocol aims to achieve this task by decentralizing the routing decisions and minimizing the required knowledge-base to make these decisions.

In this section, we start by presenting the procedure that enables us to gain full access rights on the Android device. Second, we explain how we use the wireless ad-hoc mode. Finally, we describe the steps necessary to run the B.A.T.M.A.N. daemon on the devices, and the routing logic of the protocol.

Rooting an Android phone

Most Android phones run constrained versions of the operating system, due to limitations implemented by the manufacturers and operators. In order to be able to benefit from the full potential of the Android OS, and more specifically to switch on the ad-hoc mode on the wireless interface of the devices, we need first to gain full control of the Linux subsystem. This implies modifying the access rights on the phones, by performing what is called a rooting procedure. In order to better understand this procedure, we present it in relation to the Android boot process. A more detailed description of the initial boot stages of an Android device can be found at ENEA development team [10]. The rooting procedure of an Android smartphone has been successfully described on the unlockr website [16].

The Android boot process can be described as follows:

1. When power is supplied to the device, the *Boot ROM*, which is a small hardwired piece of code, is executed. Its responsibility is to launch the Boot Loader.
2. The *Boot Loader* starts by setting up external RAM, then sets up file systems, additional memory, network support

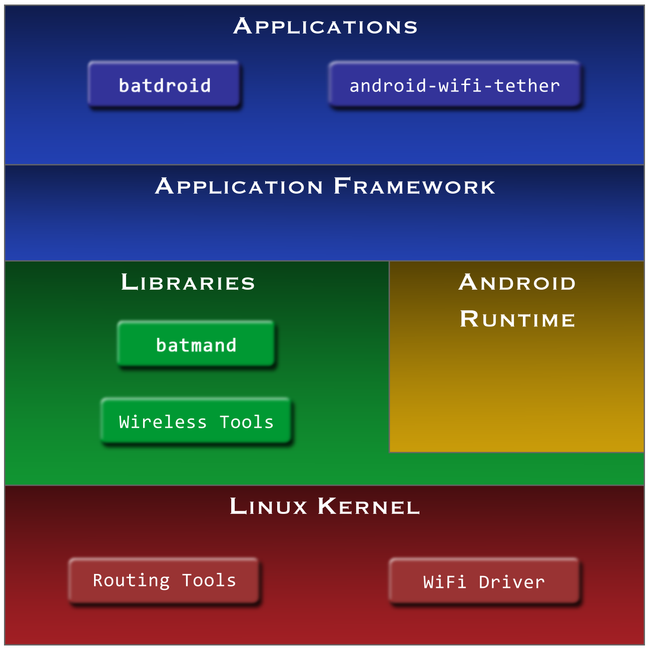

Figure 2. System design in relation to Android OS Architecture

and possibly other things such as security options. Finally, it detects available Linux kernels to boot, i.e. *boot images*, and the kernel takes over.

3. The *Linux kernel* sets up everything that is needed for the system to run. Particularly, it enables the launch of user space processes, and launches the first one, called the init process.

4. The *init process* is where all other system processes originate. Its responsibility is to launch the system service processes using the init.rc script.

5. The *Zygote* is the main instance of the *Dalvik VM*. It starts executing and initializes Dalvik [8].

6. The *System Server* starts the Android services.

7. Once the system boot has completed, the standard broadcast action called *ACTION_BOOT_COMPLETED* is sent.

Rooting an Android phone is generally done in two main steps: (1) replace the Boot Loader, (2) load a customized boot image. Both of this steps happen during the Boot Loader phase of the Android boot process. The replacement of the Boot Loader requires an update of the Boot ROM code. Both are protected by the phone's CID, which is an identification number located on the SD card. In order to circumvent this protection, it is possible to create a *goldcard* with a custom CID. Using this goldcard, we can remove the limitations on the Boot ROM code, and load a generic Boot Loader. Finally, we can have the Boot Loader load a custom boot image with enhanced features, including the superuser permission rights. See [7] for an example of such a customized Linux kernel for Android phones.

Enabling ad-hoc WiFi

Fitting a complex operating system on a hand-held device is a heavy optimization task. In the case of Android OS, it was done by using a bare-bone UNIX environment called BusyBox [29]. The tools provided by this system are highly customized and have only the necessary features. Unfortunately, some extra configuration parameters usually provided by the wireless extension tools were left out of the commercialized Android OS versions.

Mobile devices running Android OS are usually purchased together with subscriptions that include data transfer deals, allowing mobile internet access to the user. However, it is often more convenient to browse the internet on an actual computer than on a phone. The demand among users to be able to share the internet access of smartphones with other devices has caused the emergence 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.

The application that we are using is called android-wifi-tether [21]. It switches the wireless interface to ad-hoc mode and provides a user-interface for configuring the network parameters, such as ESSID, static IP address and subnetwork. The application is built upon the 'iwconfig' command from the Linux Wireless Extension tools package [27]. The package, or a subset of it, can be compiled specifically for the Android platform, using the native toolset extension [1] of the Android Software Development Kit [2]. Because the Android OS relies on a Linux kernel, and hence features the same driver model, tools without complex library dependencies work on Android phones seamlessly.

Our system uses the android-wifi-tether application to bring up the wireless interface in ad-hoc mode and to set the user defined IP address. Once the interface is available and configured properly, it can be used to initiate connections to other devices. Following is a presentation of how the B.A.T.M.A.N. daemon can be implemented and run on Android phones.

Running the B.A.T.M.A.N. daemon

Similarly to the wireless tools, batmand [4] has no library dependencies. This allows the porting procedure to Android OS to be fairly straightforward. Even though there are some minor modifications to be done, the task can be carried out given sufficient C programming experience.

The B.A.T.M.A.N. protocol is available as a user space daemon and as an experimental kernel module. Since there has already been some efforts made to use the user-space daemon, it was decided to build on the experience already gained. The project is called batdroid [17], it consists of an Android user interface and a set of libraries. The UI provides a start/stop button and possibility for configuration. The libraries are the natively compiled B.A.T.M.A.N. daemon and the aforementioned android-wifi-tether application. There are no B.A.T.M.A.N. specific settings exposed to the user.

The available configurations are android-wifi-tether specific settings, as discussed previously. When forming the mesh network, the participating devices need to be configured. The wireless interfaces must be set to the same channel and ESSID set, while their IP addresses should be different.

The routing daemon is launched by pressing a start button on the phone's screen. Once the daemon is running, we can connect to it as a client, using the shell provided by the Android Debug Bridge (ADB). In particular, the debug level 1 command allows us to observe the routing decisions. However, focusing only on these log messages is not sufficient. We have to ensure that the routing rules and tables of the system are modified accordingly. B.A.T.M.A.N. uses policy routing, which enables it to benefit from special routing functions provided by the Linux kernel. A Linux system provides many routing tables; one of them is referred to by the number 254, and generally used for standard routing operations. However, B.A.T.M.A.N. uses a different set of tables, as well as specific routing rules. Each table maintains a specific element of the mesh, as explained in [6]. This setup has the advantage of making B.A.T.M.A.N.-related routing issues distinct from the rest of the system settings, and easily accessible.

The system routing tables are modified by kernel calls using the Netlink library. For this to work, the Netlink support must be fully enabled in the kernel running on the phone. Unfortunately, the kernel configuration of our test devices did not allow for full Netlink support. Therefore the first experiments were unsuccessful. The routing decisions were observed but no communication channels could be established between the devices. The kernel call that was supposed to modify the routing tables of the system was not able to do so. This issue could have been resolved by recompiling the kernel with the appropriate configuration. However, we fixed it by implementing a work-around that manages the routing tables with the 'ip' command-line tool. It is possible to instruct the B.A.T.M.A.N. daemon to use a custom routing policy. The policy can be defined in the form of a shell script, where decisions are emitted to the standard input of the script and the shell commands can be used to carry out necessary actions. This matched perfectly our needs for the work-around. A simple script was written, to parse the decisions and turn them into valid parameters for 'ip' command calls. After a few minor modifications of the batroid application to integrate the work-around, we achieved connectivity between the devices.

The B.A.T.M.A.N. routing logic

The approach of the B.A.T.M.A.N. algorithm is to divide the knowledge about the best paths between nodes in the network to all participating nodes. Each node perceives and maintains only the information about the best next hop towards all other nodes. Therefore, the need for global topology knowledge is unnecessary.

In a B.A.T.M.A.N. network, all the participating nodes periodically broadcast originator messages (OGMs) to its neighbors. An OGM consist of an originator address, a sending node address and a unique sequence number. Upon receiving the OGMs, the neighbor nodes change the sending address to their own address and re-broadcast the message, and so on and so forth. Thus the network is flooded with originator messages. The sequence numbers are used to determine the recency of an OGM, in order to allow nodes to re-broadcast OGMs at most once.

See [4] for more information on the workings of the algorithm, and [19] for a performance comparison between B.A.T.M.A.N. and OLSR.

EVALUATION

Data gathering

Our evaluation of the routing behavior of the B.A.T.M.A.N. daemon is based upon the gathering and study of the batmand debug log files. They are automatically generated by the B.A.T.M.A.N. daemon, and stored on the devices, to be generated during routing or at a later time. There are 5 different debug levels to choose from, as explained in the batmand HowTo [28]. Typically, we are referring to level 1 logs, that shows information on the participants in the network.

The test scenarios were run on three mobile devices:

- Device A: the Linux laptop computer (IP 10.10.3.14).
- Device B: the first HTC Legend phone (IP 10.10.3.53).
- Device C: the second HTC Legend phone (IP 10.10.3.58).

In order to access the routing debug outputs via the ADB shell, we connected two accessory laptop computers to the phones as clients. Table 1 and 2 present the results of our tests. The time stamps correspond to the uptime of the B.A.T.M.A.N. daemon on the Linux computer.

Following is an explanation of the data presented in the tables:

- Timestamp: the uptime of the system.
- Device IP: the IP of the node that is running the B.A.T.M.A.N. daemon.
- Originator: the IPs which we can reach.
- Nexthop: the IPs which we send our packets to when we want to reach the Originator IP. The number in the parenthesis indicates the link quality of the connection, calculated by keeping track of the amount of Originator Messages received and their recency.
- Potential nexthops: the IPs which are one hop neighbors. The B.A.T.M.A.N. daemon will choose the router with the best link quality from the potential nexthop list.

We have seen previously that B.A.T.M.A.N. routing knowledge does not necessarily imply the ability to actually exchange packets between nodes. There can be other obstacles, such as the inability for batmand to write to the system's IP tables, as discussed previously. However, we have

Timestamp	Device IP	Originator	Nexthop	Potential nexthops
UT: 0d 1h55m	10.10.3.14	10.10.3.53	10.10.3.53 (255)	10.10.3.53 (255), 10.10.3.58 (233)
		10.10.3.58	10.10.3.58 (252)	10.10.3.58 (252), 10.10.3.53 (245)
	10.10.3.53	10.10.3.14	10.10.3.14 (243)	10.10.3.14 (243), 10.10.3.58 (219)
		10.10.3.58	10.10.3.58 (251)	10.10.3.58 (251), 10.10.3.14 (211)
	10.10.3.58	10.10.3.53	10.10.3.53 (255)	10.10.3.53 (255), 10.10.3.14 (197)
		10.10.3.14	10.10.3.14 (237)	10.10.3.14 (237), 10.10.3.53 (227)
UT: 0d 1h58m	10.10.3.14	10.10.3.53	10.10.3.58 (237)	10.10.3.53 (232), 10.10.3.58 (237)
		10.10.3.58	10.10.3.58 (249)	10.10.3.58 (249), 10.10.3.53 (0)
	10.10.3.58	10.10.3.53	10.10.3.53 (251)	10.10.3.53 (251), 10.10.3.14 (183)
		10.10.3.14	10.10.3.14 (225)	10.10.3.14 (225), 10.10.3.53 (0)
UT: 0d 2h 0m	10.10.3.14	10.10.3.58	10.10.3.58 (251)	10.10.3.58 (251)
	10.10.3.58	10.10.3.14	10.10.3.14 (220)	10.10.3.14 (220)
UT: 0d 2h 2m	10.10.3.14	10.10.3.53	10.10.3.53 (244)	10.10.3.53 (244), 10.10.3.58 (227)
		10.10.3.58	10.10.3.58 (244)	10.10.3.58 (244), 10.10.3.53 (234)
	10.10.3.53	10.10.3.14	10.10.3.58 (215)	10.10.3.14 (203), 10.10.3.58 (215)
		10.10.3.58	10.10.3.58 (254)	10.10.3.58 (254), 10.10.3.14 (182)
	10.10.3.58	10.10.3.53	10.10.3.53 (247)	10.10.3.53 (247), 10.10.3.14 (201)
		10.10.3.14	10.10.3.14 (214)	10.10.3.14 (214), 10.10.3.53 (191)

Table 1. B.A.T.M.A.N. log output: Test Scenario 1 & 2

Timestamp	Device IP	Originator	Nexthop	Potential nexthops
UT: 0d 2h31m	10.10.3.14	10.10.3.53	10.10.3.53 (242)	10.10.3.53 (242), 10.10.3.58 (137)
		10.10.3.58	10.10.3.58 (209)	10.10.3.58 (209), 10.10.3.53 (170)
	10.10.3.53	10.10.3.14	10.10.3.14 (21)	10.10.3.14 (21)
		10.10.3.58	10.10.3.14 (18)	10.10.3.14 (18)
	10.10.3.58	10.10.3.53	10.10.3.14 (54)	10.10.3.14 (54)
		10.10.3.14	10.10.3.14 (189)	10.10.3.14 (189)
UT: 0d 2h33m	10.10.3.14	10.10.3.53	10.10.3.53 (246)	10.10.3.53 (246), 10.10.3.58 (42)
		10.10.3.58	10.10.3.53 (167)	10.10.3.58 (85), 10.10.3.53 (167)
	10.10.3.53	10.10.3.14	10.10.3.14 (167)	10.10.3.14 (167), 10.10.3.58 (42)
		10.10.3.58	10.10.3.58 (182)	10.10.3.14 (80), 10.10.3.58 (182)
	10.10.3.58	10.10.3.53	10.10.3.53 (125)	10.10.3.14 (39), 10.10.3.53 (125)
		10.10.3.14	10.10.3.53 (75)	10.10.3.14 (43), 10.10.3.53 (75)

Table 2. B.A.T.M.A.N. log output: Test Scenario 3 & 4

managed to resolve the aforementioned issues, and to successfully achieve connectivity whenever running batmand. Connectivity should therefore be assumed whenever routing is mentioned in the following evaluation.

Data evaluation

(1) The Single Hop Scenario
We were able to successfully implement a fully connected mesh between the devices A, B and C. The log files show that B.A.T.M.A.N. correctly handles single hop routing, as can be seen in Table 1, timestamp `UT:1h55m`. Node A knows that the packets that are destined to B should be routed to B, and the ones that are destined to C should be routed to C. The same can be said of the routing knowledge of nodes B and C.

(2) The Join / Leave Scenario
In order to verify that it was possible for a device to dynamically join and leave the existing mesh, we turned device B off, and switched it on again. The resulting log outputs correspond to the timestamps `UT:1h58m`, `UT:2h0m` and `UT:2h2m` in Table 1. We were able to witness the disappearance and subsequent reappearance of device B on the log files of devices A and C.

(3) The Multi Hop Scenario
We deployed the devices in a multi hop network configuration, by placing the devices so that B is out of range of C, but A is in range of both, i.e. in between them. As the log files with timestamp `UT:2h31m` in Table 2 show, node B knows that node C is part of the network, and that it should route to A packets that are destined to it. The same is true for node C.

(4) The Network Reconfiguration Scenario
We achieved network reconfiguration by physically moving node C towards and past node A, then past node B, until node B was positioned between the two other devices. The log files with timestamp `UT:2h33m` presented in Table 2 show the results of the new configuration. We can notice

that devices A and C are still in range of each other, but that routing goes through device B anyway, due to its superior link quality.

Evaluation results

After performing the four described tests and analyzing the obtained experimental results, we can confirm that it is possible to form ad-hoc networks between Android smartphones, using the B.A.T.M.A.N. routing protocol. The results show that the routing logic of the B.A.T.M.A.N. daemon is correct. When performing test scenario 2, we have noticed that it takes time for batmand to remove a dead node from the routing tables. The reason for this is that batmand is specifically designed for network configurations where links are unreliable, and where the failure of a link is often temporary, while the actual node is still alive. The results of test scenario 4 confirm that the nodes choose their best nexthop option depending on link quality. In this case, all nodes were at one-hop distance of each other, but node B being well positioned in between the other two, it was correctly elected by the two other nodes as bridging node.

DISCUSSION

Suggested improvements

Our system is based on the use of the already existing batdroid application, which itself is using the existing android-wifi-tether application and the B.A.T.M.A.N. daemon. However, there are issues concerning the integration of batdroid and android-wifi-tether to the Android framework. Currently, these applications cannot work appropriately if both of them are installed on the phone. We suggest establishing communication between the applications by using Intents and the BroadcastReceiver in the Android framework.

The use of Netlink to modify the kernel routing tables would be an optimal setup and a way to save battery life. We see reconfiguring and recompiling the kernel as a necessary task. Once at the kernel level, it would make sense to enable the experimental kernel module of B.A.T.M.A.N. protocol called batman-adv [5], which is the main focus of the current development efforts. Batman-adv uses the same algorithm, but performs routing on layer 2 (Ethernet layer). Some of the advantages are: (1) independence from the network-layer, (2) nodes do not need to have an IP address, (3) lower energy consumption and optimized data flow by avoiding processing overhead.

As a way of summarizing the suggested improvements, we envision a new version of the android-wifi-tether application that would provide the option to use B.A.T.M.A.N. advanced layer 2 routing. The hope is that batman-adv will reduce battery consumption, which remains the main obstacle towards convenient usage of intensive networking protocols such as B.A.T.M.A.N..

Towards free telephony

B.A.T.M.A.N. is only a means towards an end. To the end user, it is only a setting providing connectivity, like switching on the WiFi on a laptop. The end goal here is to implement an application that provides telephony services. Off-the-shelf applications that are meant to use mesh network protocols are rare. The ones that are available are mostly related to file-sharing, which is the main application area that peer-to-peer networking has been used for. Applications using VoIP (Voice over Internet Protocol) are available, but they are very resource-intensive, due to their centralized design. They are often based on client-server architectures, where the server component is rather complex, maintaining the knowledge of the network structure and executing resource-intensive computations. Every user needs to log on to the server in order to access the service. This scenario is not imaginable in an ad-hoc network, where there might be thousands of nodes without Internet access. It is possible to port such client-server applications to mesh networks, by running both components on the same device, but the suitability of this approach is highly dependent on the resource exhaustion of the server. Disregarding the structure of the underlying mesh network can lead to unnecessary computations, hence wasted resource consumption (e.g. battery life).

VoIP applications provide the user with free internet telephony. They can be used on Android phones, provided an Internet access point. Their features include free calls to other VoIP users and cheap rates for calling land-line and mobile phones. The newest Android OS release, version 2.3 Gingerbread, comes with a SIP stack and provided APIs which makes the development of VoIP applications extremely simple. In this context, such applications are multiplying, and we see Mobile Telephony Service Providers evolve to slowly become Mobile Internet Service Providers. However, despite common belief, VoIP is not a synonym for free telephony. Using a VoIP application is not free from Service providers that are likely to charge the user, either for calls or for data transfer. Telephony can only be free if it builds upon a self-maintaining network structure.

Technically, a decentralized VoIP application can be developed. The application logic should support a smart way to search contacts and a convenient security policy to avoid annoyances. We believe that developing such an application is the next step towards free ad-hoc telephony.

CONCLUSION

In this paper, we have presented the necessary steps for successfully establishing an ad-hoc network of mobile phones. We have done this by porting the B.A.T.M.A.N. routing protocol on HTC Legend phones running version 2.1 of the Android OS. We have shown how to gain full control of the Linux subsystem, how to enable the ad-hoc wireless mode, and how to run the B.A.T.M.A.N. daemon. We have evaluated our system and demonstrated that B.A.T.M.A.N. behaves correctly on Android OS. We have described the similarities between Unix/Linux systems and the Android OS and demonstrated that we can use Linux development knowledge to extend the features of Android OS. Additionally, we have suggested some improvements of our system, in an effort to obtain computationally cheap routing for ad-hoc mesh network. Finally, we have hinted at the possibilities to adopt VoIP to decentralized networks, thus achieving free ad-hoc telephony.

Our belief is that the kernel space implementation of the B.A.T.M.A.N. routing protocol will eventually become a standard kernel module and that developing a truly decentralized VoIP application will be rendered easy on the latest and upcoming versions of Android OS. These factors suggest that smartphones will soon become modern walkie-talkies, allowing users to make free calls. However, resource constraints remain in the way, as well as the manufacturers's strategy to withhold control of the required kernel features.

ACKNOWLEDGMENTS

We would like to thank Antoine van Gelder, for sharing the batdroid application, and for his help in overcoming the initial technical difficulties. We are grateful to PITLab Manager Sebastian Büttrich for supervising our work, as well as providing us with the necessary hardware. We would like to thank Pervasive Computing Course Manager Jakob E. Bardram, as well as lecturers Arne John Glenstrup, Aurelien Tabard and Francesco Zanitti, for their teachings and support.

REFERENCES

1. Android Native Development Tools. `http://developer.android.com/sdk/ndk/`.
2. Android SDK. `http://developer.android.com/sdk/`.
3. Asterisk: Open Source Communications. `http://www.asterisk.org/`.
4. B.A.T.M.A.N. / Open Mesh. `http://www.open-mesh.org/`.
5. B.A.T.M.A.N. advanced. `http://www.open-mesh.org/wiki/batman-adv`.
6. B.A.T.M.A.N. Getting behind the routing vodoo. `http://www.open-mesh.org/wiki/RoutingVodoo`.
7. Cyanogenmod, custom ROM for Android OS. `http://www.cyanogenmod.com`.
8. Dalvik Virtual Machine. `http://www.dalvikvm.com/`.
9. Documentation of Village Telco Workshops. `http://wiki.villagetelco.org/index.php?title=Village_Telco_Documentation`.
10. ENEA Android Competence Center. `http://www.enea.com/Templates/Product_____36678.aspx`.
11. Linksys WRT54G series. `http://en.wikipedia.org/wiki/Linksys_WRT54G_series`.
12. OpenWrt: Wireless Freedom. `http://openwrt.org/`.
13. Serval: uniting the world through communication. `http://www.servalproject.org/`.
14. Terranet. `http://www.terranet.se`.
15. Village Telco: an easy-to-use, scalable, standards-based, wireless, local, do-it-yourself, telephone company toolkit. `http://www.villagetelco.org/`.
16. D. Cogen. How to: Root the android htc legend. `http://theunlockr.com/2010/06/07/how-to-root-the-htc-legend/`, June 2010.
17. A. V. Gelder. B.A.T.D.R.O.I.D. connects your rooted Android handset to B.A.T.M.A.N. mesh networks a.k.a. android-batdroid. `http://code.google.com/p/android-batdroid/`.
18. P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks. In *Multi Topic Conference, IEEE INMIC 2001. Technology for the 21st Century. Proc. IEEE International*, pages 62–68, 2001.
19. D. L. Johnson, C. Aichele, and N. Ntlatlapa. A simple pragmatic approach to mesh routing using batman. In *Wireless Communications and Information Technology in Developing Countries (WCITD'2008). 2nd IFIP International Symposium on*, 2008.
20. D. L. Johnson and D. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, 353:153–181, 1996.
21. H. Mue, U. Ada, B. Buxton, and A. Robinson. Wireless Tether for Root Users a.k.a. android-wifi-tether. `http://code.google.com/p/android-wifi-tether/`.
22. A. Neumann, C. Aichele, and M. Lindner. Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.). Technical report, IETF Network Working Group, Apr 2008.
23. C. Perkins and E. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, volume 2, pages 90–100, 1999.
24. J. Rosenberg, dynamicsoft, H. Schulzrinne, C. U., G. Camarillo, Ericsson, A. Johnston, WorldCom, J. Peterson, Neustar, R. Sparks, dynamicsoft, M. Handley, ICIR, E. Schooler, and AT&T. SIP: Session Initiation Protocol. Technical report, IETF Network Working Group, June 2002.
25. D. Rowe. The mesh potato. *Linux J.*, 2009, December 2009.
26. P. G. Stephen. Serval-DNA: The Serval Project's Distributed Numbering Architecture (DNA). `http://code.google.com/p/serval-dna/`.
27. J. Tourrilhes. Wireless Tools for Linux. Technical report, Hewlett Packard Laboratories, August 2000.
28. W. Tsai. *B.A.T.M.A.N. Daemon HowTo*, January 2009.
29. D. Vlasenko. BusyBox: The Swiss Army Knife of Embedded Linux. `http://www.busybox.net/`.