University of Trento,
Italy

Laboratory activities for the Nomadic Communications course 2009/2010: Report 2

■	Marco Dalla Torre	136311	marco.dallatorre@studenti.unitn.it	■
■	Simone Raffaele	140950	simone.raffaele@studenti.unitn.it	■

Anno Accademico 2009/2010

Abstract

This report describes the second experimental activity done for the course of Nomadic Communications. We can divide it in three parts: while the first part is introductory, presenting the work and the testbed configuration used to take the measurements, the remaining two parts are about the actual experimental results. In particular, the second part is a theoretical approach to the experiments performed, while the third part analyzes the experiments' results. The objective of the experiments is to analyze the behaviour of the *B.A.T.M.A.N.* and the *O.L.S.R.* protocol implementations. Both these parts will try to approach the problem first from a theoretical point of view and then with the analysis of the actual measured values.

Contents

1 Introduction

The goal of the experimental activity described here is to analyze and understand the behaviour and capabilities of the main implementations of two of the most prominent 802.11 routing protocols for mesh networks, *B.A.T.M.A.N.* and *O.L.S.R.*. The respective implementations we use are those offered at *batmand* [2] and *olsrd* [5] projects homepages.

1.1 Testbed description and setup

All the measurements have been done in the faculty in the rooms 106, 107 and 201, using mainly the CISCA provided equipment. For our group this consisted in:

- four laptops with identical hardware and with Ubuntu Linux installed.
- a fifth laptop, with the same hardware configuration as those just described, running the Backtrack Linux live cd[1] has been used to perform live packet capture of the experiments using TCPDUMP[6]. These data has been analyzed by using Wireshark[7].

More specifically, the laptops are equipped with the *Intel PRO/Wireless 2200BG* integrated wireless card, capable of supporting the b and g communication protocols, and also the *monitor* operational mode.

Network set-up: The ad-hoc network is set-up by executing on each station the commands in the next listing:

```
sudo ifconfig eth3 down
sudo ifconfig eth3 192.168.13.67 netmask 255.255.255.0 broadcast 192.168.13.255
sudo iwconfig eth3 mode ad-hoc essid "Noi" channel 1 rate 54M
sudo ifconfig eth3 up
```

In order to avoid the groups to interfere with each other as much as possible, all the groups' ad-hoc networks are configured to use the three non overlapping frequencies allowed by 802.11: we used that on channel 1 (2.412MHz). The above lines of course, differ for each station in the IP assignments. During the experiments we kept the following statically assigned addresses configuration:

- ip address: 192.168.13.65, mac address: 00:13:ce:da:89:a3 as source
- ip address: 192.168.13.66, mac address: 00:13:ce:da:2b:fa as first hop

[1]homepage: http://www.backtrack-linux.org/

- ip address: 192.168.13.68, mac address: 00:13:ce:d8:b8:ea as second hop or alternative hop, depending on the test
- ip address: 192.168.13.67, mac address: 00:13:ce:da:94:59 as sink

Available routes set-up: Since the test stations are all in the same room sitting next to each other, in order to test the routing protocols in a plausible configuration we simulate the absence of physical communication link between pairs of laptops by using the the `iptables` program suite. This is possible because the implementations we are using all try to find out available routes through UDP communications. Just blocking these communications works because both implementations operate by manipulating the system's routing tables on the basis of the discovered routes. We blocked some selected links by using the `iptables` command like in the following:

```
iptables -F
iptables -A INPUT -s 192.168.13.67 --protocol UDP --source-port 698 -j DROP
iptables -A INPUT -s 192.168.13.68 --protocol UDP --source-port 698 -j DROP
```

For example, by inserting the above lines at the 192.168.13.65 station's root terminal we are telling the ip stack to ignore incoming UDP packets from the two specified addresses (192.168.13.67 and 192.168.13.68) having source port 698 (in this case port 698 is used by the *O.L.S.R.* protocol).

```
iptables -F
iptables -A INPUT -s 192.168.13.65 --protocol UDP --source-port 1966 -j DROP
```

With these lines instead we are blocking the *B.A.T.M.A.N.* discovery protocol from address 192.168.13.65. As before, the resulting effect is that the blocked network interface will not try to directly communicate with the blocking one, but instead will try to rely on indirect paths.
With similar lines on all the test stations we can effectively set up a network where packets are routed through multiple hops. All the computers involved in the test can then communicate with each other through predefined 802.11 wireless paths set up by the meshing protocols.

A brief consideration about *B.A.T.M.A.N.*: its newest kernel-space implementation (the `batman-advanced` branch) is no longer relaying on layer 3 to discover new routes, making the trick just presented ineffective and requiring instead a more complex solution. However, this does not affect us since the version installed with the provided Ubuntu is the older user-space implementation. While the new implementation brings a number of advantages (e.g. lower cpu usage and independence from layer 3 protocols to name a few)[1], the principles behind the protocol should roughly remain the same, so we maintain that even using the older version can result in an up-to-date study of the protocol's working principles.

Protocols set-up: As soon as the network is finally configured and running, we start the protocols. The line for *olsrd* is:

```
/usr/sbin/olsrd -i eth3
```

while for *batmand* the line is:

```
/usr/sbin/batmand -d 1 eth3
```

Benchmarking tools: The benchmarking tools chosen for measuring the connection throughput are `iperf`[3] and `netperf`[4].

1.2 Experiments description

In the experiments reported here we consider mainly two type of results:

- the performance from the point of view of the receiver given one or two hops
- the route set-up time in response to various simulated underlying topology changes

All the experiments were performed in the university's new IRST building. This building is well known to be not "wireless friendly", being rich of both wireless interference sources and attenuation inducing materials. Unluckily we have no other places to use to perform the tests in.

2 Speed Study

In this section we measure the transfer rate through a number hops managed by the two routing protocols. In particular we have arranged for the following topology configurations:

- a one hop configuration, with three stations connected

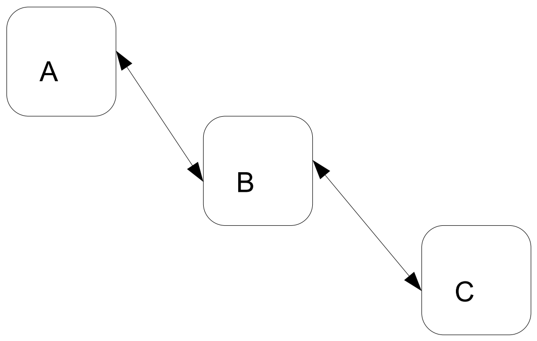

Figure 1: 1 hop single path topology

- a one hop configuration with parallel routes, with four stations connected, two of them being a possible path to reach the sink

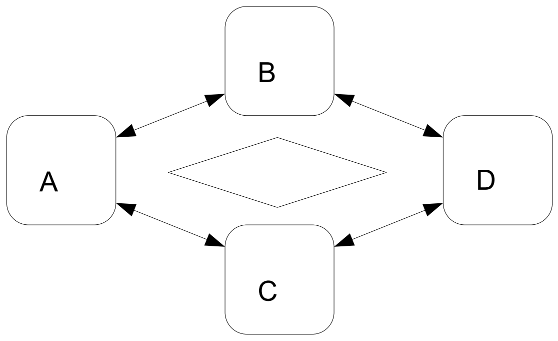

Figure 2: 1 hop parallel topology

- a two hops configuration, with four stations connected

In an initial setup of the tests we did also distinguish between *free* and *fixed network rate* (set through the `iwconfig` tool), only to discard this distinction (leaving the parameter unset) for the simple reason that this setting does not

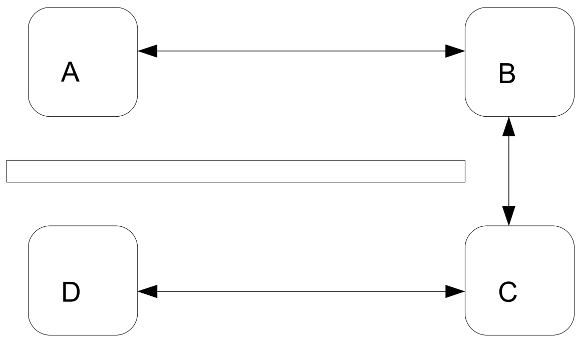

Figure 3: 2 hops topology

seem to make much sense in ad-hoc networking. This is because `iwconfig` only allows us to see and change a logical representation of the connection to the ad-hoc network. Newest tools like `iw`, that unfortunately we could not use, allow to review and modify parameters for each connection to the neighboring participants. This makes much more sense as every connection may have its totally different set of characteristics. Both *batmand* and *olsrd* have then been tested with these configurations, in this section we will proceed to present the results.

olsrd	`Single Path`	`Parallel Path`
`Average Packet Loss`	16.94%	2.73%
`Average (Mbit/sec)`	6.58	2.72
`Max (Mbit/sec)`	10.4	5.18
`Min (Mbit/sec)`	3.35	1.24
`Std Deviation (with no TX error)`	2.16 (3.42)	2.3 (1.52)
batmand	`Single Path`	`Parallel Path`
`Average Packet Loss`	0.01%	3.59%
`Average (Mbit/sec)`	3.53	4.9
`Max (Mbit/sec)`	7.57	8.53
`Min (Mbit/sec)`	1.23	1.78
`Std Deviation (with no TX error)`	2.3 (2.34)	2.75 (3.06)

Table 1: results of the experiments in the 1 hop configurations using *olsrd* and *batmand* with UDP, for single and parallel paths. Average, Max and Min values refer only to 0% data loss runs. Standard deviation values are listed both considering all the 22 tests' average speeds and, enclosed in parenthesis, only those with no error.

2.1 Throughput with 1 hop

In the following measurements we did put up a two hops configuration, in both single and parallel path versions. The test was originally done using `iperf` with the following parameters:

```
/usr/bin/iperf -c192.168.13.65 -b54M -i5 -t20
```

where, of particular interest:

- -b 54M forces to send data at 54 Megabits/second
- -i 5 sets the interval in seconds between bandwidth reports
- -t 20 sets the time in seconds to transmit for

The experiment is repeated 22 times, for a total of 440 seconds worth of transmission using *olsrd*. Unfortunately the results for these tests are not very meaningful: because of UDP and the output rate we are forcing on the sender side, a lot of the sent data are not received by the server. Additionally, the values reported at each interval (omitted in this report) are those sent through the first link, not considering at all the data rate arriving from the hop the sink. All we can save from those experiments are the (few) measurements for the 20 seconds test reported by the server where no packet got lost. Results from that data are summarized in table 1: as expected it is difficult to draw any conclusions from these tests. The packet loss amount and the observed speed are highly variable, especially in the single path case. The parallel path seems to be more prone to comparison. Still, we reckon the difference in speed has more to do with disturbance on the channel than with the routing protocol in use.

After noticing the problem caused by the presence of packet loss, we decided then to change the experiments using a more reliable configuration, this time using `netperf` in TCP mode. This is the starting command:

```
/usr/bin/netperf -H192.168.13.65 -l20
```

This time too, each experiment was 20 seconds long, but it was repeated only 10 times because of time constraints. With this configuration even if TCP adds overhead to the communication, we are able to use the reported rates without fear of using values belonging to an unsuccessful transmission. The resulting plots are showed in figures 4 and 5, while a summary of the key values is present in table 2. From the values it is apparent that the rates are in general quite low compared with some of the values obtained in the previous tests. Still, the speed is slightly higher compared with the cases where less losses happen. The difference in speed cannot only be explained with the replacement of UDP with TCP: other possible causes may be that transmission speed is no more forced to 54Mbit/sec by the

	batmand		*olsrd*	
	Single	Parallel	Single	Parallel
Average	3.26	3.22	3.31	3.39
Max	3.49	3.45	3.67	3.73
Min	2.91	2.95	2.89	2.93
Std Deviation	0.17	0.16	0.24	0.23

Table 2: Aggregates of the experiments results in the 1 hop configurations

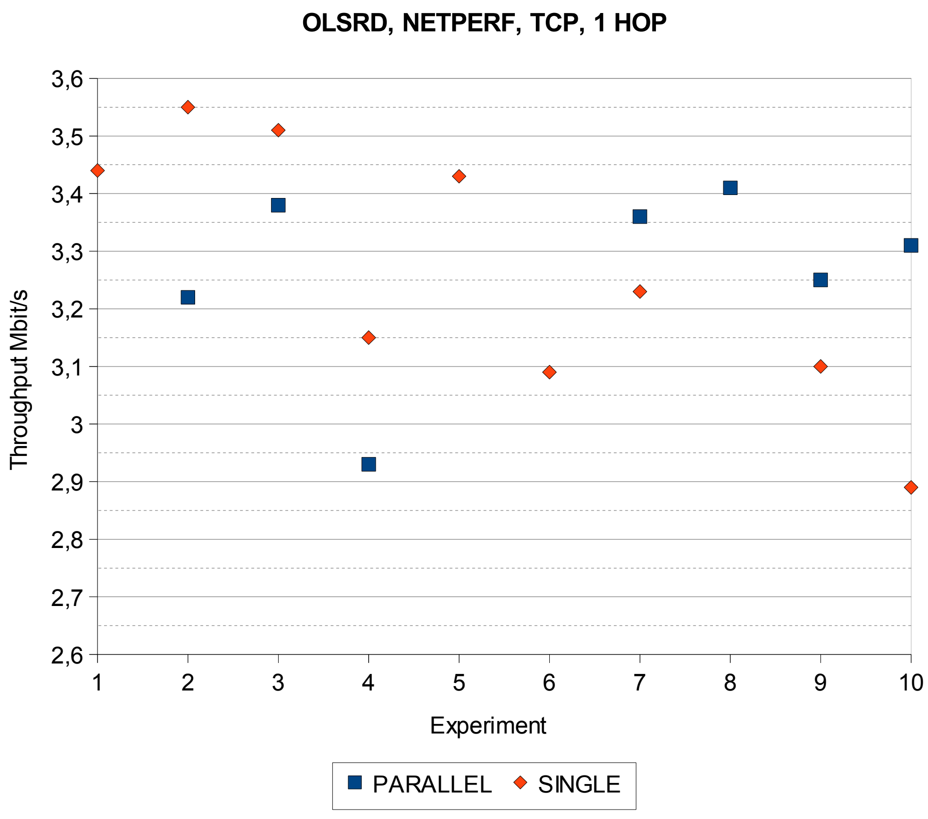

Figure 4

sending application (which could also explain the much lower standard deviation) and that the tests were done at a different date (which could lead to a different level of channel pollution). Other than this, we can add that it does not seem that the routing protocol is much meaningful to the actual transmission rates. Clearly, given how they both work by just manipulating the kernel routing table, the only way in which they can lead to different results in throughput is by generating a different amount of traffic. But by just looking at these values, while *olsrd* seems

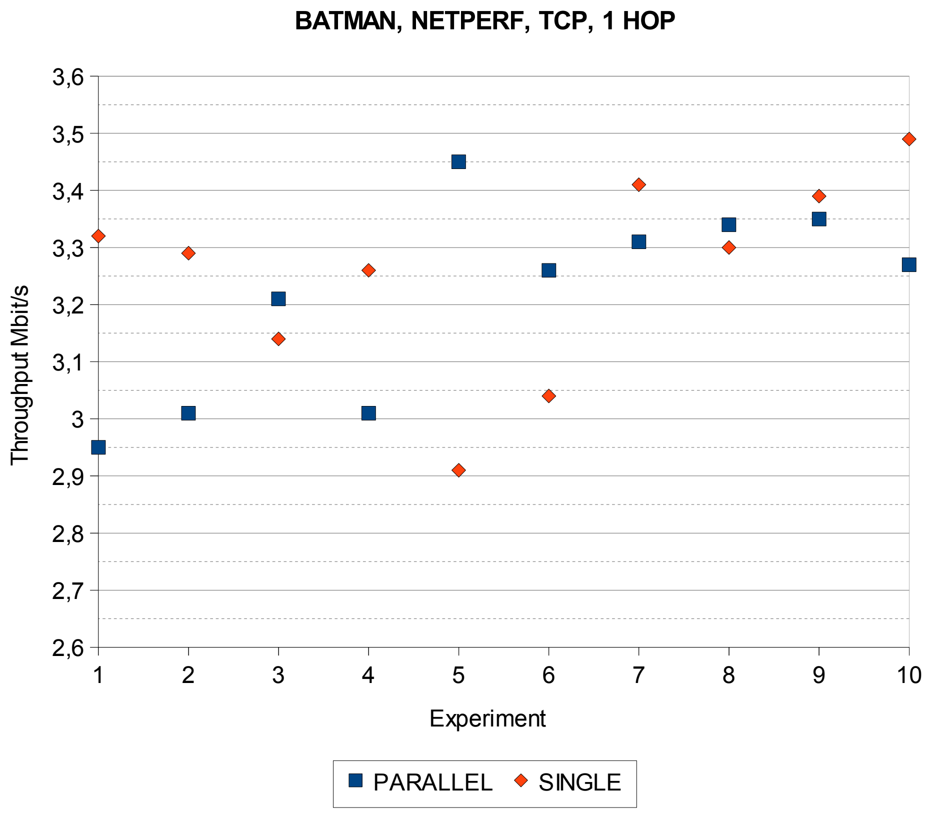

Figure 5

to have a bit of a lead over *batmand* by in the average throughput values, the differences are small enough to not lend themselves to clear conclusions. And once again, the high quantity of interferences can have had some influence.

2.2 Throughput with 2 hops

Now we present the results from the test using the 2 hops topology. Even in this case, the tests are performed using the `netperf` tool. Results are shown in figure

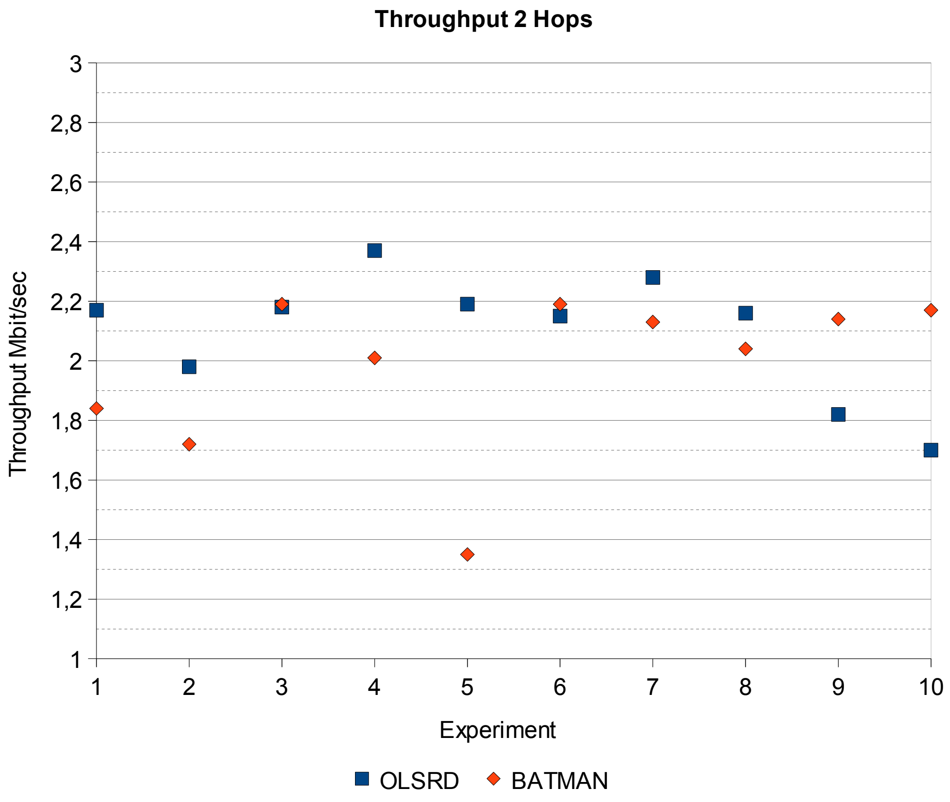

Figure 6

6 and summarized in table 3. Like in the 1 hop case, we do not observe particular differences by using the two routing protocols, but the *O.L.S.R.* implementation seems to come out with a slight lead over *batmand*. These result seems to be coherent with the results obtained in section 2.1, although we again want to stress the fact that the margins are very tiny and that as always results can be heavily influenced by the channel condition. For example, by eliminating the *batmand* lowest value reported (1.35 Mbit/sec) we have an overall sequence of values that are much in line with those reported by *olsrd*, resulting in an average of 2.05 Mbit/sec much similar to the value of 2.1 we have with the *O.L.S.R.* implementation.

	batmand	*olsrd*
`Average (Mbit/sec)`	1.98	2.1
`Max (Mbit/sec)`	2.19	2.37
`Min (Mbit/sec)`	1.35	1.7
`Std Deviation`	0.26	0.2

Table 3: results of the experiments in the 2 hops configuration using *olsrd* and *batmand* with `netperf` in TCP.

2.3 Comparison throughput with direct, 1 hop and 2 hops

In this part we briefly try to compare the scalability of the two protocols, that is, we examine eventual performance losses related to the growth of network participants. To do so, we reuse the values obtained by 10 runs of the `netperf` tool already used in the previous sections. In figure 7 we have a graphic comparison of the

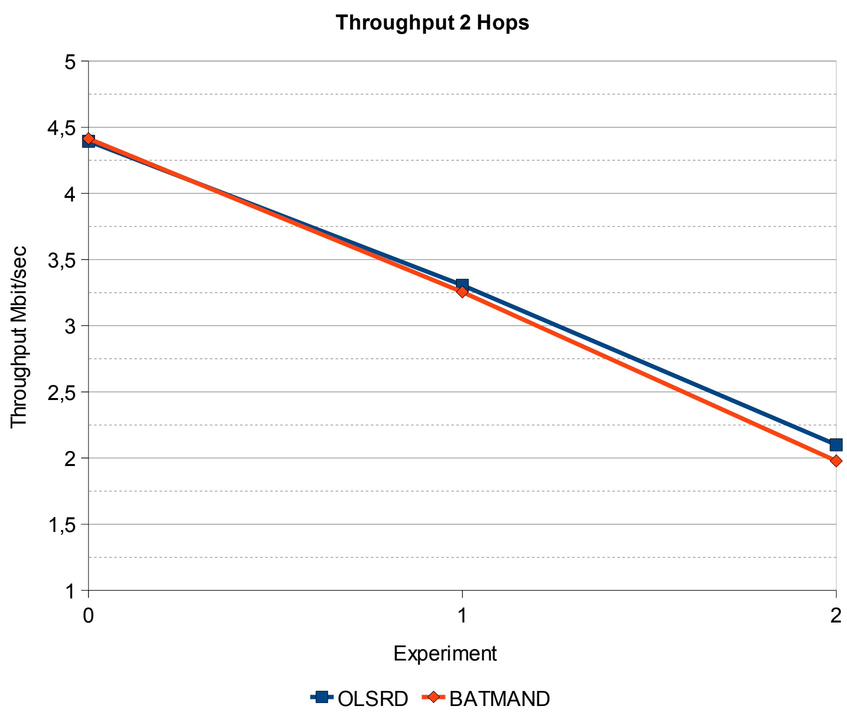

Figure 7: Comparison of the *O.L.S.R.* and *B.A.T.M.A.N.* protocols implementations on average throughput using TCP at 0, 1 and 2 hops

two protocols. As it is readily apparent, the are no significant variations in the average values to report. For the sake of completeness we reported the values on table 4. Unfortunately, we could only use a maximum of four stations to do our

N hops	*batmand*		*olsrd*	
0	4.41	`degradation`	4.39	`degradation`
1	3.26	74%	3.31	75%
2	1.98	60%	2.1	63%

Table 4: results of the experiments' throughput average in Mbit/sec with 0, 1 and 2 hops configuration using *olsrd* and *batmand* with `netperf` in TCP. The `degradation` columns are intended as the throughput's degradation with respect to the configuration immediately above

tests, while these protocols are supposed to work with a number of nodes in the hundreds. We believe that having such a small testbed is not enough "hard work" for them to be push at their limit, hence having the possibility to observe what is the better approach.

3 Convergence Study

In this section we will measure the responsiveness of the routing protocol in reaction to changes to the physical topology. Such changes can be a sudden link disappearance, some interferences that cause the link to be unusable, etc… Figure 8 shows the general idea beyond our study. To do the measurements we have

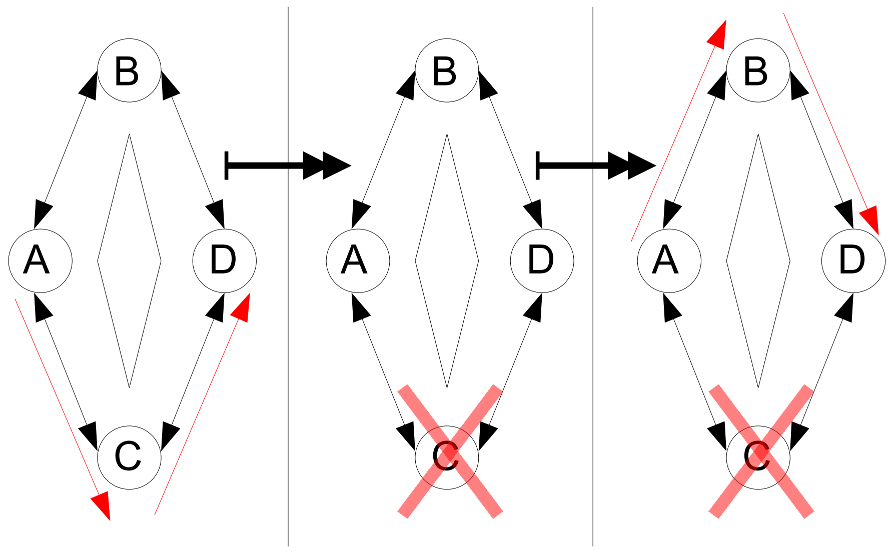

Figure 8

decided to use the following configuration of iperf:

```
/usr/bin/iperf -c192.168.13.65 -b54M -i1 -t30
```

This instruction will be repeated 10 times for each test, during which at some point, one of the two possible data path, the one actually in use, will be forcefully turned "dead" by disabling the wireless card of its node. We have observed that, following the "death" of a node, the iperf client returns, for some time, a set of data that shows that the transmission is not in act: these data can be a series of 0s or "impossible" data (values above the 15 - 20 Mb/s). The data obtained from iperf are then compared with the data registered by Wireshark (that is running on the fifth laptop). This comparison was useful to understand that the iperf data produced by the client laptop were actually meaningful.

For simplicity of representation we will transform the data graphs as shown in figure 9: in the simplification each of the throughput data during the transmission will be set at a constant value of 1, only to show that there is a transmission in act, while the data during the time the transmission is not in act, will be set to 0. This can be done because our interest is in the number of seconds during which there is no transmission instead of the actual throughput. The wireshark

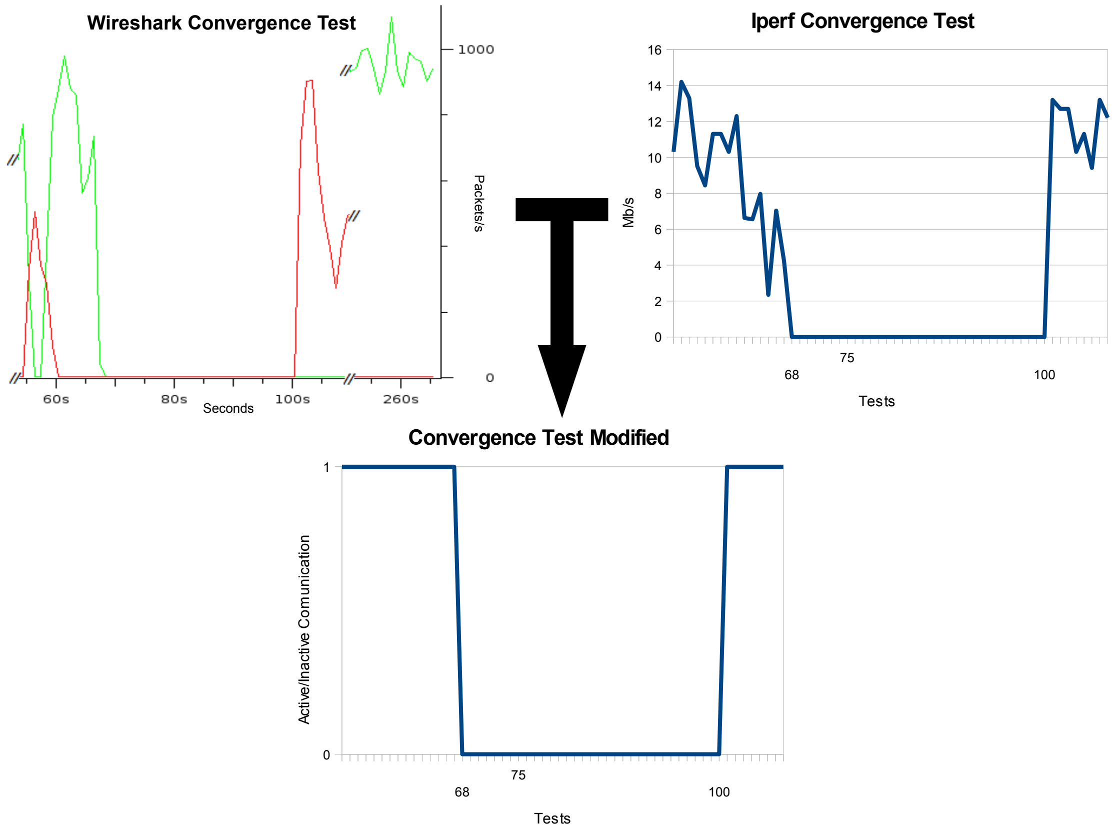

Figure 9

generated graph as seen in the image presents two distinct colored lines. These represent iperf transmissions on the two different last hop segments.

We will show the results obtained from 3 tests for each of the 2 Mesh protocols implementations used, *olsrd* and *batmand*. During the first two tests we have done the following: after some time from the start of the test we have turned off the wireless card of the path used for the transmission of the stream of data, then, when the protocol has reacted to the "death" of the node and changed the path, we have turned on the wireless card of the "dead" node. By doing this we have had the opportunity to put to "death" another node during a single test, measuring two data concerning the convergence time of the protocols. The third test is a bit different. In fact we have set one of the two intermediate nodes as a "lossy" node: this is done with the help of iptables, limiting the number of protocol control packets that the node accepts. These are the iptables instructions used for *olsrd*:

```
sudo iptables -A INPUT -p udp --dport 698 -m limit --limit 360/min -j ACCEPT
sudo iptables -A INPUT -p udp --dport 698 -j DROP
```

while the following is the one used for *batmand*:

```
sudo iptables -A INPUT -p udp --dport 1966 -m limit --limit 360/min -j ACCEPT
sudo iptables -A INPUT -p udp --dport 1966 -j DROP
```

3.1 Convergence Time

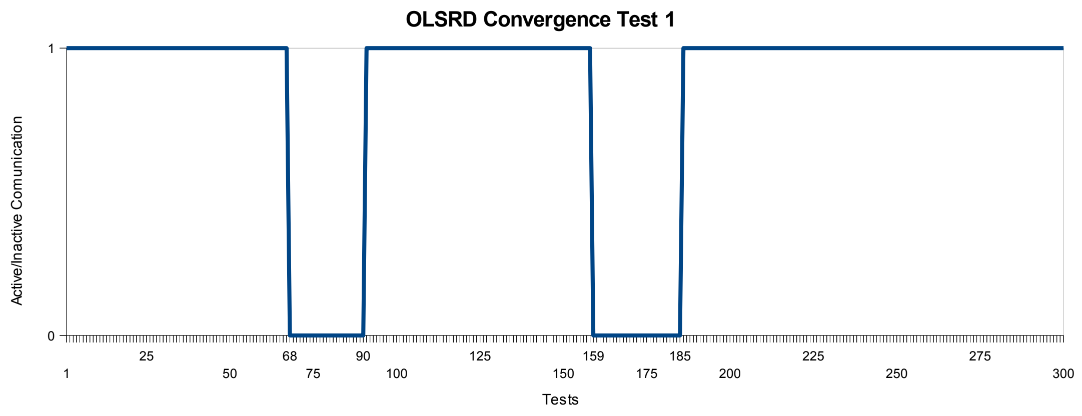

Figure 10

Figures 10 and 11 show the tests results obtained from *olsrd*. In the first graphic we have the convergence times from second 68 to 90 and from second 159 to the 185, while in the second the times are from second 68 to 100 and from second 165 to 187. This means that we have 4 measurements of: 23, 27, 32, and 22 seconds, with a mean value of 26 seconds.

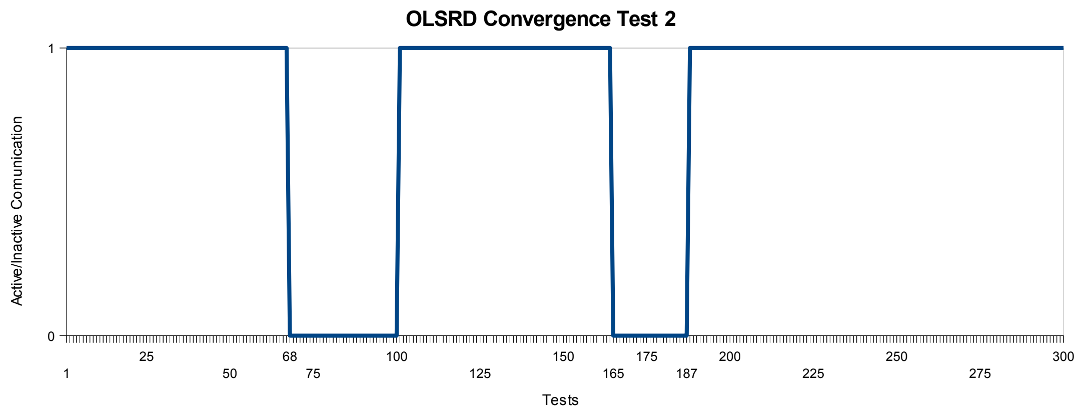

Figure 11

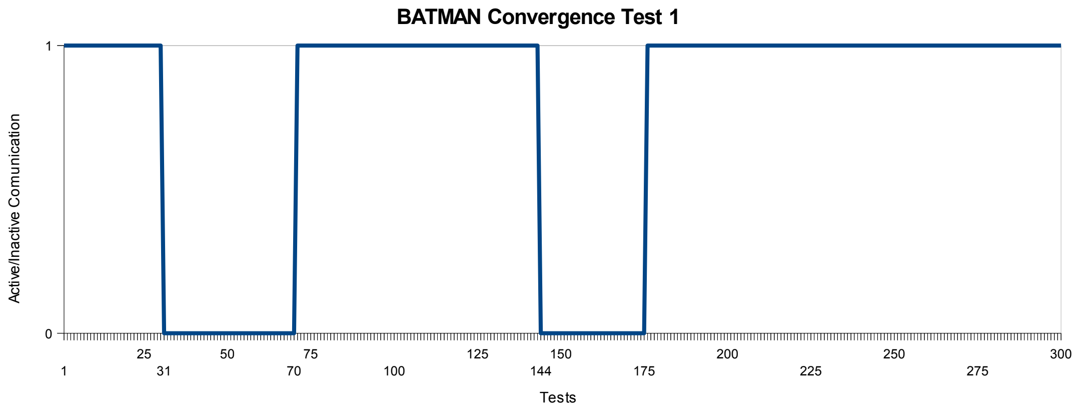

Figure 12

Figures 12 and 13 show, instead, the tests results obtained from *batmand*. Again we have 4 convergence times: in the first test from second 31 to 70 and from 144 to 175, from the second test from second 35 to 60 and from 141 to 190. These measurements leads to downtime measurements that are 40, 32, 26 and 50 seconds long, with a mean value of 37 seconds. Although the data obtained are not so different it seams that *batmand* is a little bit slower than *olsrd* in reacting to a suddenly missing node.

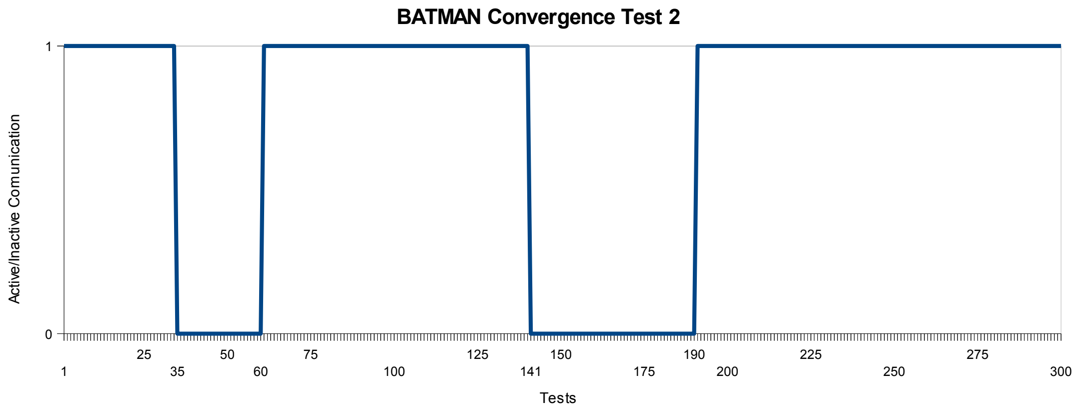

Figure 13

3.2 Convergence Time with Lossy Node

To conclude with this section we show the two tests done with the "lossy" path:

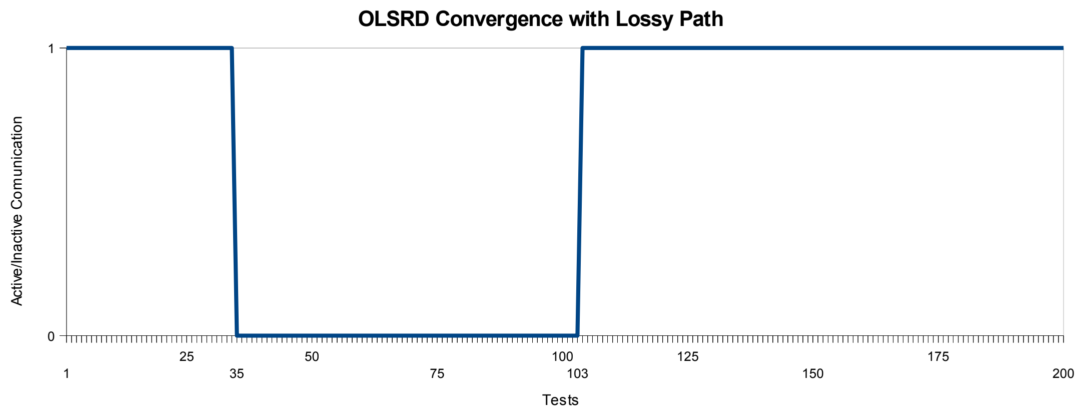

Figure 14

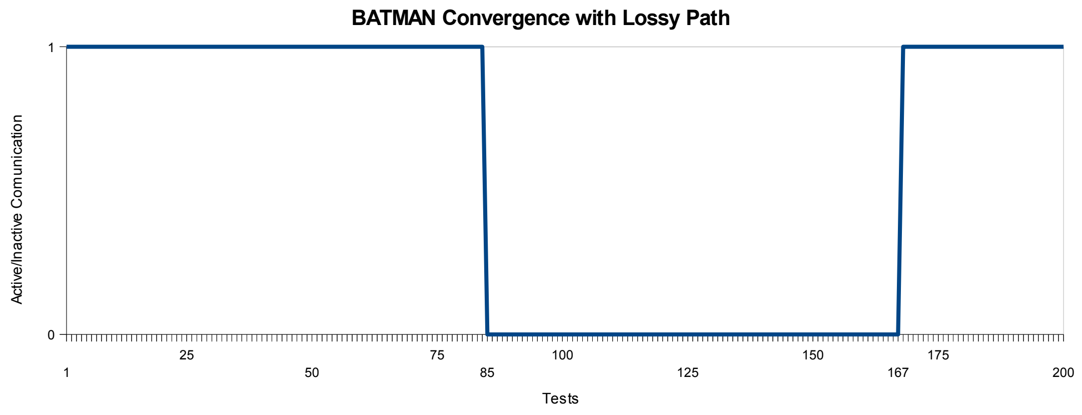

Figure 15

Figures 14 and 15 shows the results obtained with both protocols: for *olsrd* we have a convergence time of 69 seconds (from second 35 to 103) while for *batmand* we have a convergence time of 83 seconds (from second 85 to 167). Again, even if the difference is small, it seems that *batmand* is slower in it's reactions.

3.3 Convergence Results

Although the results obtained seems to tell us that *olsrd* is faster than *batmand* most of the times, we have also to consider that the environment in which we have done our tests is not ideal for direct comparisons. Also we are using only a small number of nodes, while the protocols are designed to work on a much larger scale (hundreds to thousands of nodes). Table 5 shows the various obtained results in a compact way.

	Times (in seconds)			
	`Test 1`	`Test 2`	`Average`	`Lossy Test`
olsrd	23, 27	32, 22	26	69
batmand	40, 32	26, 50	37	83

Table 5: Results of the various tests about convergence time of both OLSR and B.A.T.M.A.N

4 Conclusions

During the work for this report we have had the opportunity to work with two of the most prominent proactive protocol implementations for *wireless ad-hoc mesh networks*, *batmand* and *olsrd*. In our work we have analyzed the behavior of the two protocols in a restricted ambient, with particular attention for their throughput and convergence performances. The actual results obtained seem to be telling us that in the case of pure throughput there is a nearly equal behavior, while in the case of response to the "death" of a node (Convergence), *olsrd* seems to be a little faster. These conclusions nonetheless have to be taken with a grain of salt, since the test are made in conditions and modalities which we believe are far from the typical field of operation for wireless mesh networks. The reasons are multiple: the very limited number of participating stations and the interference busy location in which the network operated are the main concerns. Also the "tricks" used to simulate unreachable and-or bad link nodes may have created some unexpected behavior with use of both the mesh protocols and the benchmarking tools we have utilized for capturing the results. Instead it was actually interesting to observe the different responses of different protocols to the same kind of problems.

References

[1] B.a.t.m.a.n. advanced
http://www.open-mesh.org/wiki/batman-adv.

[2] B.a.t.m.a.n. (better approach to mobile ad-hoc networking)
http://www.open-mesh.org.

[3] Iperf homepage
http://sourceforge.net/projects/iperf/.

[4] Netperf homepage
http://www.netperf.org.

[5] O.l.s.r. homepage
http://www.olsr.org.

[6] Tcpdump homepage
http://www.tcpdump.org/.

[7] Wireshark homepage
http://www.wireshark.org/.