

# PERFORMANCE AND FLEXIBILITY OF OPEN SOURCE ROUTING SOFTWARE <sup>1</sup>

V. Eramo, M. Listanti, A. Cianfrani, E. Cipollone

University of Roma “La Sapienza”, INFOCOM Dept.

**Abstract-** Routing protocols are a critical component in IP networks. Beside dedicated hardware, a great interest on routing systems based on open software is raising among Internet Service Providers. Many open source implementations of this protocol have been developed, among which Quagga and Xorp are the most used in PC-based router. In this paper we evaluate the OSPF performance of Quagga and Xorp routing software according to the test methodologies defined within the Internet Engineering Task Force. Moreover we describe a set of changes made on Quagga code in order to optimize some processes, whose algorithms were not efficient. In order to show the flexibility of an Open Source Routing Software, we have also implemented in Quagga an incremental algorithm for the evaluation of the shortest path. The realized implementation allows the shortest path computation time to be reduced of about the 97%.

**Index terms:** Open Source Code, Quagga and Xorp Routing Software, Open Shortest Path First, Dijkstra’s algorithm, Binary Heap, Incremental Algorithm.

## I. Introduction

The diffusion of open software implementing routing protocols, together with the big computing power of normal PCs, have been raising a big interest towards the possibility of developing a complete routing system based on open source software and standard low-cost hardware [1,2]. What still lacks is to verify if and when actually the performance of these PC-based-routers can be compared to that of commercial systems. We have

---

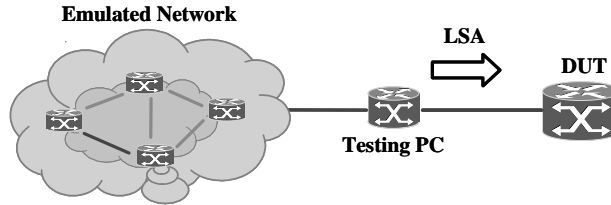
<sup>1</sup> This work has been supported by MIUR (Ministero dell’Università e della Ricerca) through the BORA-BORA project

developed a set of tests to analyze the performance of a router running the OSPF protocol [3], according to the IETF specifications [4]. In particular we have taken Black Box measures [5] of the time needed to perform the Shortest Path First (*SPF*) computation on a Personal Computer (PC) equipped with operating system Linux and Quagga [6] or Xorp (eXtensible Open Router Platform) [7] routing software. The evaluation of the *SPF* computation time raises the evidence of a lack of optimization in the Dijkstra's algorithm implemented in Quagga. A deep analysis of the code evidenced that the data structure used to implement the Candidate List during the *SPF* calculation was not optimized. So we have modified the code implementing a binary heap data structure [8]. In our previous work [9] a similar result has been obtained evaluating the *switching time*, a performance index of data plane. In order to show the flexibility of an open source software, we have implemented in Quagga an incremental algorithm.

The organization of the paper is as follows. The test-bed for the *SPF* time evaluation is illustrated in Section II. The OSPF performance of Quagga and Xorp is shown and compared to the one of the Cisco 2801 router in Section III. The optimization of Quagga is described in Section IV. The performance of the incremental algorithm implemented in Quagga is evaluated in Section V. Our main conclusions are discussed in Section VI.

## II. Test-Bed for the Evaluation of the *SPF* Computation Time

The test aims at determining how long it takes for a Device Under Test (DUT) to complete the *SPF* computation. The DUT can be either a market router or a PC based router. The test configuration used is reported in Fig. 1. The network topology is made up of two real routers (a testing PC and the DUT) and a variable number of fictitious routers and networks, so that the DUT will have to find the shortest path to all the vertexes of the emulated network, a vertex being either a network or a router. The testing PC is running a C++ software allowing: i) any network topology to be generated; ii) Link State Advertisements (LSA) describing the network topology to be generated and sent to the DUT.

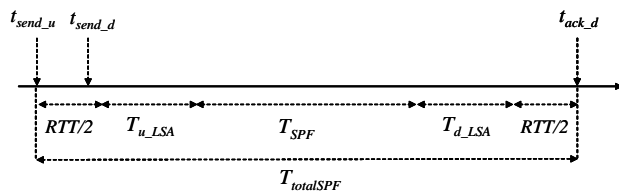


**Fig. 1.** Test-bed for the evaluation of the *SPF* computation time.

The IETF has been defining the measure methodology for the *SPF* computation [4]. Next we describe the procedure allowing the *SPF* to be computed. To understand the test methodology proposed, we remember that OSPF routers use to schedule the instant in which the *SPF* computation starts to avoid to perform the calculation too many times when receiving Update LSAs [3]. So, when an Update LSA arrives, notifying for example a cost variation in an emulated network link, the *SPF* computation start time is scheduled with a fixed delay, a timer is set and the *SPF* calculation starts only when the timer expires. Moreover, another timer enforces a lag between two consecutive *SPF* computations. In particular the following two timers are defined in [3]:

- *spf\_delay*: time between receiving an Update LSA and starting the *SPF* computation;
- *spf\_hold\_time*: time between two consecutive *SPF* computations.

The *SPF* computation time measurement consists of two different steps, with different settings of these two timers. First as shown in Fig. 2 we set both the timers to 0, so forcing the DUT to immediately start the *SPF* computation when it receives an Update LSA. In Fig. 2 we denote with *RTT* the Round Trip Time and further we assume that the propagation time is the same for the two directions Testing PC-DUT and DUT-Testing PC.



**Fig. 2.** Time measure  $T_{totalSPF}$

The first step of the test consists in loading the emulated network into the DUT and in sending an Update LSA at time  $t_{send\_u}$  followed after a little delay by a Duplicate LSA at time  $t_{send\_d}$ . The DUT processes the Update LSA in the time interval  $T_{u\_LSA}$  and starts to execute the *SPF* algorithm.

Once begun, the *SPF* process cannot be interrupted, and goes on till its end. Then the DUT processes the Duplicate LSA in the time interval  $T_{d\_LSA}$  and sends back immediately, according to the OSPF protocol rules, its Acknowledge LSA. Thus we can use the Acknowledge LSA of the Duplicate LSA to understand when the *SPF* computation ends. In particular in this first test we measure the time  $T_{totalSPF}$ , which represents the time difference between the sending of the Update LSA at time  $t_{send\_u}$  and the receiving of the Acknowledge LSA of the Duplicate LSA at time  $t_{ack\_d}$ . As shown in Fig. 2, the  $T_{totalSPF}$  time can be expressed as follows:

$$T_{totalSPF} = RTT + T_{u\_LSA} + T_{SPF} + T_{d\_LSA} = T_{overhead} + T_{SPF} \quad (2.1)$$

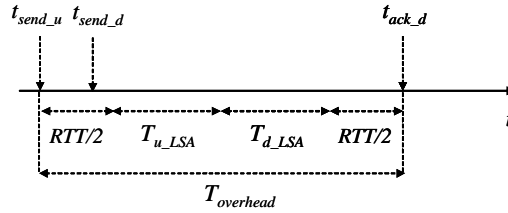
wherein

- $RTT$  is the Round Trip Time between the Testing PC and the DUT;
- $T_{u\_LSA}$  is the Update LSA processing time;
- $T_{d\_LSA}$  is the Duplicate LSA processing time;
- $T_{SPF}$  is the *SPF* computation time;
- $T_{overhead} \equiv RTT + T_{u\_LSA} + T_{d\_LSA}$ .

Hence in the performed measure we are able to evaluate  $T_{totalSPF}$  but we are interested in evaluating  $T_{SPF}$ . In order to make this we evaluate  $T_{overhead}$  and subtract it from  $T_{totalSPF}$  obtaining  $T_{SPF}$ , that is:

$$T_{SPF} = T_{totalSPF} - T_{overhead} \quad (2.2)$$

We estimate  $T_{overhead}$  with a second test where we set both the *SPF*  $spf\_delay$  and  $spf\_hold\_time$  timers to high values (60 s).



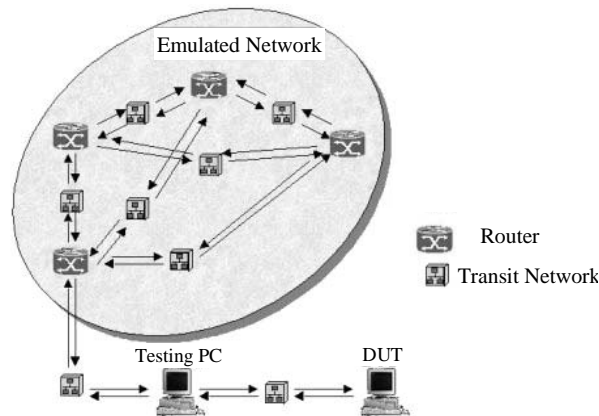
**Fig. 3.** Time measure  $T_{overhead}$ .

The DUT receives the Update LSA and schedules the *SPF* computation start time but does not execute it because the timers are high. The DUT processes the Duplicate LSA and sends back the Acknowledge LSA of the Duplicate LSA as illustrated in Fig. 3. The time difference between the sending of the Update LSA and the receiving of the Duplicate LSA Acknowledge is exactly  $T_{overhead}$ .

### III. OSPF Performance of Quagga and Xorp Open Source Routing Software

All performed tests are based on fully meshed network topologies, with each router connected to each other through a different transit network. Fig. 4 shows an example of fully meshed topology with 4 routers. It is important to remark that in representing the emulated network as a directed weighted graph [3], each router and each transit network becomes a vertex of the graph, and each network-router link becomes an edge. Each edge is labelled with a cost representing the interface cost of the link connecting a router to a network [3]. In the following the cost of all the edges will be chosen to be equal. When the Update LSA is sent as mentioned in Section II, an interface cost is varied so that the *SPF* computation procedure is primed.

The *SPF* computation complexity will depend on the number of vertexes and edges in the graph. Now let us denote with  $N$ ,  $M$  the number of vertexes and the number of edges of the graph.



**Fig. 4.** The emulated network topology considered in the test-bed is fully meshed.

If we consider an emulated network topology composed by  $R$  routers, we have that:

$$N = \frac{R(R-1)}{2} + R \quad (3.1)$$

$$M = 2R(R-1) + 2 \quad (3.2)$$

The number of edges  $M$  is proportional to the number of vertexes  $N$ , in particular from (3.1) and (3.2) we can assume that  $M=O(N)$ .

Experimental values taken on a PC based router and on a Cisco 2801 access router are reported in Fig. 5. We report the *SPF* computation time as a function of the number of network topology vertexes. The PC used is equipped with a 2.4Ghz processor, a 512Mbyte memory and Quagga 0.98 or Xorp 1.2 routing software. Notice as the experimental values taken on a Cisco 2801 router perfectly agree with the trend foreseen by the Dijkstra's algorithm. In fact as shown in Fig. 5 we notice as the experimental measure curve of the Cisco 2801 fits the curve  $0.005 \cdot N \cdot \log N$  very well. On the contrary the results obtained on a router based on the PC hardware and equipped with Quagga or Xorp routing software are quite different. Of the two Open Source Routing Software, Xorp performs much better than Quagga. For example when the number of vertexes is 5000, the *SPF* time in Xorp is about 6 s. On the contrary the *SPF* time in Quagga increases up to 16 s as the number of vertexes reaches 5000 and measured values fit on the  $6.06 \cdot 10^{-4} \cdot N^2$  interpolating curve, as shown in Fig. 5.

On the basis of these results we retain that some changes are needed inside the Quagga 0.98 code, to obtain performances comparable to commercial routers. Next section is dedicated to Quagga optimization.

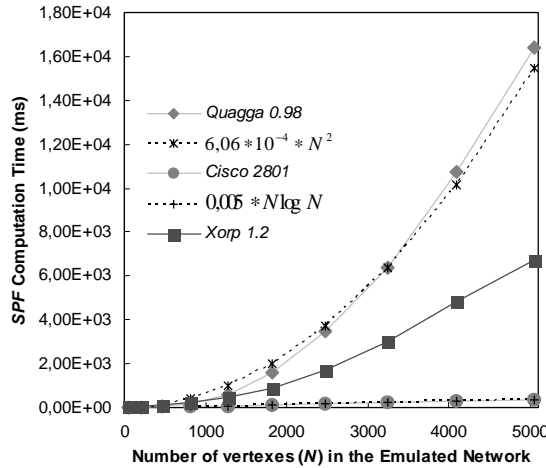


Fig. 5. The *SPF* Computation Time in a Cisco 2801 router and a PC based router.

#### IV. Optimization of the *SPF* Computation Time in Quagga

The *SPF* computation is based on the Dijkstra's algorithm, as described in [3]. The algorithm examines the directed weighted graph already described

in Section II, in order to find the shortest paths from a root vertex to each other vertex in the graph. All these paths give rise to a Spanning Tree of the graph. In Quagga 0.98 the directed graph is itself represented by the LSA set, stored in the LSA database. During the iterations of the algorithm all the vertexes must be extracted, one by one, from the graph and inserted into the Spanning Tree. Moreover Quagga 0.98 also uses the Candidate List, a structure that contains all the reachable vertexes that have not yet been inserted but can be reached from vertexes already inserted into Spanning Tree. The Candidate List is used as a step in the middle during the migration of the vertexes from the graph to the Spanning Tree. Each of the reached vertexes is extracted from the graph, inserted into the Candidate List and provided with a key that represents the total cost needed to reach it starting from the root and crossing the minimum cost path composed by only the vertexes that have already been inserted into the Spanning Tree. According to the Dijkstra's algorithm a vertex will be extracted from the Candidate List and inserted into the Spanning Tree only when it becomes the node with the lowest key in the Candidate List. The algorithm finishes when all of the vertexes have been inserted in the Spanning Tree and that occurs when the Candidate List becomes empty. During this procedure the Candidate List is the most stressed structure. Its management is the key point of the resulting global performances, and it is performed by four different functions: the *Extract-Min*, the *Insert*, the *Decrease-Key* and the *Lookup* functions.

In Quagga 0.98, the Candidate List is implemented with a linked list, whose elements are stored in key increasing order. It is possible to prove that in this case the Dijkstra's algorithm complexity is  $O(N^2+NM)$ .

Next we propose an optimization of the *SPF* algorithm. In Section IV.1 a Binary Heap data structure implementing the Candidate List will be proposed and its complexity will be evaluated. Because a Binary Heap data structure does not support the *Lookup* function efficiently, in Section IV.2 we will illustrate how this operation can be eliminated by modifying the LSA database data structure. Finally the results concerning the modified Quagga will be evaluated in Section IV.3.

#### IV.1 A Binary Heap to Implement the Candidate List

We have modified the Quagga original version with a *patch* available in [10]. In the new Quagga version we have chosen the binary heap data structure to replace the sorted list used in the original version.

A binary heap is a complete and balanced binary tree with a local sorting [8]. Leaves are always inserted starting from the left, and a new level is

actually created only when the previous one is complete. Thus the heap depth is always less than  $\log N$ , where  $N$  is the number of nodes. Each node of the heap has a key, and the whole heap is locally ordered on these keys, so that each node has a key lower than the ones of both its children. This particular sorting ensures that the node with the minimum key is the root of the heap. The management of the tree is based on two internal functions: the sift-up and the sift-down functions. The sift-up function brings up a node with a low key toward the root node. The sift-down function, on the other side, pushes down a node with an high key toward the leaves of the heap. These two procedures are exploited in order to perform the three main functions supported by the heap structure: the *Insert*, the *Extract-Min* and the *Decrease-Key* functions. The *Insert* function takes the new node and inserts it at the end of the heap, i.e. makes it a leaf. Then it executes a sift-up procedure on the inserted node, and brings it up to its correct position. Notice that because the *Insert* function needs at most  $\log(N)$  sift-up operations, equal to the maximum depth of the heap, its cost is  $O(\log N)$ . The *Extract-Min* function removes from the heap the node with the lowest key. This node is always at the root of the heap, and its extraction is almost costless. After this extraction we have two subtree, that are themselves ordered heap, and we need to fuse them into one single heap. To achieve this result, the *Extract-Min* function takes the last leaf of the heap and puts it at the root position, then executes the sift-down procedure on it and pushes it down to its correct position. Notice that because the *Extract-Min* function needs the execution of at most  $\log N$  sift-down operations, its complexity is  $O(\log N)$ . Finally the *Decrease-Key* function changes the key of a particular node to a lower value. Once the key value have been decreased, it executes the sift-up procedure on the node, and takes it to its new position. Notice that to realize the *Decrease-Key* function, a number of sift-up operations at most equal to the maximum depth of the heap is performed. For these reasons the *Decrease-Key* function cost is  $O(\log N)$ . The *Insert* function and the *Extract-Min* function will be performed exactly  $N$  times. The number of times in which the *Decrease-Key* function is performed depends on the values of the costs of the links. However it will be always less than the number of edges, so we can assume that the *Decrease-Key* function will be performed  $O(M)$  times. So the amortized costs are  $O(N \cdot \log N)$  for the *Insert* function,  $O(N \cdot \log N)$  for the *Extract-Min* function and  $O(M \log N)$  for the *Decrease-Key* function.

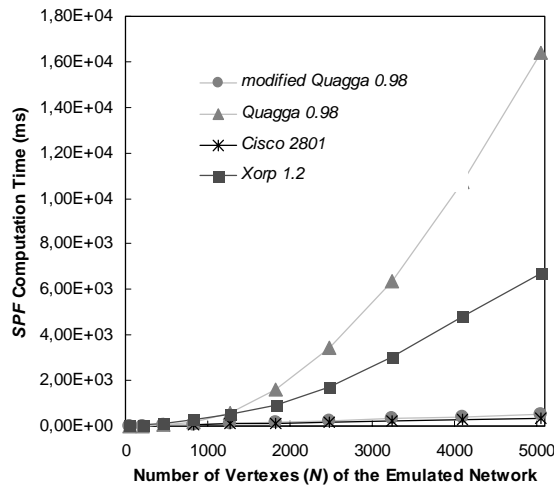
Finally, because in Section IV.2 we will show that by modifying the LSA database data structure the Lookup function is no more needed, the total cost of the new implementation of the Dijkstra's algorithm in modified Quagga becomes as expected  $O((M+N)\log N)$ . In particular when  $M=O(N)$  the amortized cost reduces to  $O(N \cdot \log N)$ .

## IV.2 The new Lookup Operation in the Candidate List

Unfortunately the changes made to implement the Candidate List rise a new problem: the binary heap need to scan one by one all the nodes to perform the *Lookup* function, as the structure is only locally ordered, thus obtaining again a  $O(N)$  cost. We have modified the LSA database data structure so that the *Lookup* function becomes no longer needed at all. In particular for each LSA, stored in the database, we have added an information denoting if or not the LSA is in the Candidate List. In positive case the information also denotes the position in the Candidate List where the vertex associated to the LSA is stored. That allows a vertex associated to an LSA to be immediately accessed during the execution of the Dijkstra's algorithm. Further, because the sift-up and the sift-down operations may change the position of a vertex in the Candidate List, a pointer to the information of the associated LSA is added for each vertex.

## IV.3 Numerical Results for Modified Quagga 0.98

The test evaluating the *SPF* time on the modified Quagga 0.98 version produced experimental results that perfectly reflect the  $N \log N$  trend. The measured values, varying the number of vertexes in the graph, are presented in Fig. 6, and compared with the same measure taken on the Cisco 2801, on the original Quagga 0.98 and on Xorp 1.2.

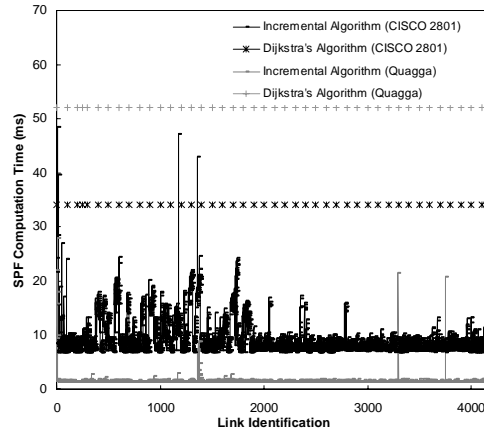


**Fig. 6.** Performance comparison among Cisco 2801, Quagga 0.98, modified Quagga 0.98 and Xorp 1.2.

It is important to note that the *SPF* time on the modified Quagga version is always less than the time needed on the original version, proving that the optimization have been successfully done. The obtained results are also lower than the ones in Xorp and only slightly worse than in Cisco 2801. Results on Xorp can be explained after a code analysis of the software: the weak point is LSA database, implemented in a sub-optimal way.

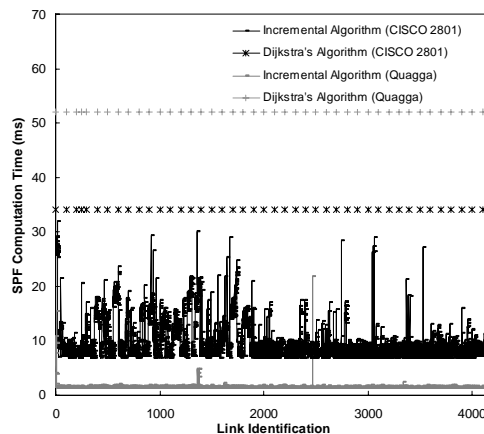
## V. An Incremental Algorithm in Quagga

When a topological change occurs in a network, the computation of the *SPF* may consume a considerable amount of CPU time. In literature have been proposed incremental algorithms [11-14] that make use of the *SPF* previously computed and update it according to the network topology changes. We have implemented in Quagga an incremental algorithm inherited from the one proposed in [12]. In particular it has been modified to take into account multipath routes to a destination node and it has been optimized for the most frequent topological changes of Internet, link failure or restoration. The incremental algorithm implemented in Quagga has been compared to the one of Cisco 2801. The comparison has been carried out by emulating on the DUT real network topologies measured within the Rocketfuel project [15]. In particular we have considered the topology of Verio, an USA Internet Service Provider whose network is composed by 893 routers and 4150 links. The *SPF* time in an incremental algorithm depends on link position and on type of change occurring, so we have chosen to measure this time when the failure and the restoration of each link of Verio occur. The *SPF* time is reported in Figs. 7,8 in the case of failure and restoration respectively as a function of the link interested. In the figures we also report the time that the DUT takes to run the Dijkstra's algorithm, that is independent of the link position. Notice as in Quagga the *SPF* time is lower when the incremental algorithm is enabled. In particular the time to run the Dijkstra's algorithm is 53 ms; while the average time is 1.5 ms in the incremental version. That leads to save about 97% in processing time. In the measure reported in Figs. 7,8 it is possible to observe some peaks, for example in the case of restoration of the links 6 and 7 the *SPF* time is 27,8 ms. and 18 ms. respectively. The peaks concern links that in the shortest path tree are near the root node and are connected to a node having many descendents, so the incremental algorithm has to perform more operations.



**Fig. 7.** Performance comparison in the case of link failure.

Observing Figs. 7,8 you can notice that though Cisco 2801 takes a time lower than modified Quagga to run Dijkstra's algorithm, in particular 34 ms. against 52 ms., the incremental algorithm implemented in Quagga is more efficient than the one implemented in Cisco 2801. In fact the average *SPF* time is 9,5 ms. in Cisco 2801 and 1.5 ms in Quagga. The incremental algorithm of Cisco 2801 allows only a saving of about 72% in processing time with respect to Dijkstra's algorithm. The measure performed on Cisco 2801 shows that there are some links whose failure causes an *SPF* time higher than the one obtained with Dijkstra's algorithm. For example in Fig. 7, the failure of link 1206 causes an *SPF* time equal to 48 ms. against the 34 ms of Dijkstra.



**Fig. 8.** Performance comparison in the case of link restoration.

## VI. Conclusions

The aim of our work was to evaluate the OSPF performance of Quagga 0.98 and Xorp 1.2 Open Source Routing Software. A test-bed has been realized and *SPF* computation time has been measured.

The experimental results for the *SPF* time obtained on a PC running Quagga have not been good. An optimization of the Dijkstra's algorithm has been made to the Quagga code by implementing the Candidate List with a Binary Heap and by modifying the data structure of the LSA database. The measure of the *SPF* time on the modified version of Quagga are slightly worse than the one of the Cisco 2801 router.

In order to show the flexibility of an open source routing software, we have implemented in Quagga an incremental algorithm for the shortest path computation. The realized implementation performs well and allows *SPF* time lower than in Cisco 2801 to be obtained. Performance and flexibility of an open source routing software raise great expectations about the possibility of making a competitive router from a standard PC.

## References

- [1] M Deval, H Khosravi, R Muralidhar, S Ahmed, S Bakshi, R Yavatkar, "Distibuted Control Plane Architecture for Networks Elements", Intel Technology Journal, Vol. 7, Issue 4, November 2003
- [2] A Bianco, J M Finocchietto, G Galante, M Mellia and F Neri, "Open Source PC-Based Software Routers: A Viable Approach to High Performance Packet Switching", in Proc of the third International Workshop, QoS-IP 2005, Catania, Italy February 2005
- [3] J Moy, "OSPF Version 2" , Request for Comments 2328, April 1998
- [4] V Manral, R White, A Shaikh, "Benchmarking Basic OSPF Single Router Control Plane Convergence", RFC 4061, April 005
- [5] A Shaikh and A Greenberg, "Experience in Black-box OSPF Measurements," in Proc ACM SIGCOMM Internet Measurement Workshop (IMW) 2001, pp. 113-125, November 2001
- [6] GNU "Quagga." [Online]. Available <http://www.quagga.net>
- [7] "Xorp." [Online]. Available <http://www.xorp.org>
- [8] A V Goldberg and R E Tarjan, "Expected performance of Dijkstra's Shortest Path algorithm", Technical Report 96-062, NEC Research Institute, Princeton, NJ, June 1996
- [9] V Eramo, M Listanti, A Cianfrani, "Switching time measurement and optimization issues in Gnu Quagga routing software", in IEEE Globecom 2005 , St. Louis, November 2005

- 
- [10] Routing Software *Patch* GNU Quagga 0.98, [http://net.infocom.uniroma1.it/projects/progetti\\_dip/zebra/index.htm](http://net.infocom.uniroma1.it/projects/progetti_dip/zebra/index.htm)
  - [11] J McQuillan, I Richer, and E Rosen, “The new routing algorithm for the ARPANET”, *IEEE Trans. on Communication.*, vol. 28, pp. 711-719, 1980
  - [12] P Narvaez, K-Y Siu, and H-Y Tzeng, “New dynamic algorithms for shortest path tree computation”, *IEEE Transaction on Networking*, vol. 8, pp. 734-746, 2000
  - [13] G Ramalingam and T Reps, “An incremental algorithm for a generalization of the shortest-path problem,” *Journal of Algorithms*, vol. 21, pp. 267–305, 1996
  - [14] D Frigioni, A Marchetti-Spaccamela, and U Nanni, “Fully dynamic algorithms for maintaining shortest paths trees”, *Journal of Algorithms*, vol. 34, pp. 351-381, 2000
  - [15] ”Rocketfuel Project.” [Online]. Available <http://www.cs.washington.edu/research/networking/rocketfuel/>