LCM, A LOAD BALANCE ALGORITHM IN MPLS-TE[1]

J. M. Arco, A. García, E. Castro y J. A. Carral
{jmarco, antonio, eva, jac}@aut.uah.es
Computer Engineering Department – University of Alcalá

Abstract

Today MPLS-TE is one of the hot technologies which every major carrier has selected for its network infrastructure. Some of the key reasons for the high level of interest in MPLS-TE include the requirements of critical network applications in the areas of link capacity and network connectivity. MPLS-TE is able to handle various paths along different routes to balance the traffic load between two points of the network. In previous works we have developed a dynamic load balance algorithm based on the maximum and mean load thresholds defined for each LSP. This paper describes the research results obtained by our load balance algorithm (LCM). In order to evaluate this algorithm, both a MPLS Linux lab test bed and a simulator have been used. Observed results show that our algorithm is able to attain dynamic load balance while avoiding undesirable oscillations.

KEYWORDS: MPLS-TE, load balance, evaluation, test bed implementation

1. Introduction

Nowadays multimedia broadband applications are demanding high-value network services. The "best-effort" broadband Internet needs to be managed more efficiently to deal with the expected traffic growth due to the multimedia broadband applications.

In the current IP networks, routing is performed by link-state protocols such as Open Shortest Path First (OSPF) [1]. In order to prevent load oscillations, these protocols use metrics that do not take into account the current traffic load. Furthermore, they find the shortest route between any pair of routers and discard other possible paths which could be used as alternative paths in order to balance IP traffic. In that way, the IP traffic is concentrated on the shortest route and network congestions may occur in some point of this route.

The continuously growing demand of available bandwidth has forced network carriers to increase both the link capacity and the network connectivity. Routing protocols should transparently balance traffic across multiple links in order to reduce network congestion and improve performance.

Multi-Protocol Label Switching Traffic Engineering (MPLS-TE) [2] is one of the hot technologies which every major carrier has selected for its network infrastructure. MPLS is based in a different paradigm, "routing at the edge and switching in the core".

[1] This work has been funded by the University de Alcalá under the "Servicios Avanzados de Red Privada Virtual de Nivel 2" ("Level 2 Virtual Private Advanced Network Services"), contract UAH-PI2005/077.

The ingress-edge MPLS router (Label Switch Router, LSR) can open several tunnels between a source and a destination and dynamically balance the load between them to prevent network congestion and improve the overall performance [3][4][5][6].

MPLS tunnels use the shortest path routes computed by classical routing protocols but may in turn use other available routes to operate alternative tunnels. The dynamic load balance, based on the current network load, autonomously distributes traffic from one congested tunnel to another tunnel with lower traffic load. This method combined with dynamic path setup protocols would enable the ingress-edge LSR to find an optimum route when the network becomes congested.

Some previous works in the field present load balance algorithms but suffer from the same oscillations problems that prevented the classical IP routing to use the current load information as an input variable [7][8].

Last year we published a new load balance algorithm named "Load balance with Congestion and Mean utilization thresholds (LCM)" [9]. Now we have deployed a MPLS Linux lab test bed and developed a software simulator which has helped us to refine its behaviour and to test the algorithm in depth.

This paper describes the research results obtained by our load balance algorithm (LCM) and explains some changes in the algorithm that improve its performace. Observed results show that our algorithm is able to attain dynamic load balance while avoiding undesirable oscillations.

The rest of the article is structured as follows. Sections 2 and 3 present the load balance approach and the proposed LCM load balance algorithm. Sections 4 and 5 show the test bed and simulation results. Finally, the last section summarises the conclusions of the work and points out some lines of work currently under development.

2. Dynamic load balance approach

A MPLS network consists of ingress-edge, core and egress-edge LSRs connected by Label Switched Paths (LSPs), figure 1.

Our load balance method could distribute IP flows between two or more LSPs in accordance to the network's traffic situation thus preventing network congestion and improving performance. The system comprises traffic statistics collection and notification functions and IP flow distribution function. Other possibilities not explored in this article can be dynamic route finding and path set up functions. The core LSRs performs the traffic statistics collection and notification functions, while the ingress-edge LSRs perform the dynamic load balance function.

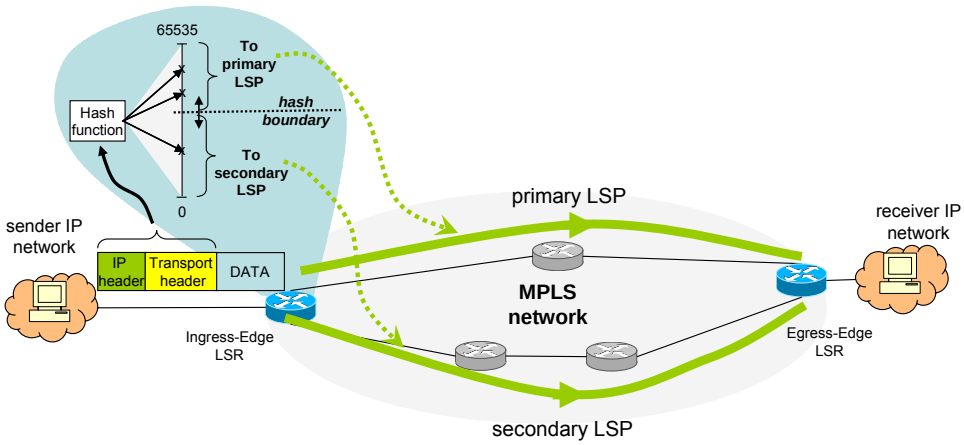

Figure 1. MPLS-TE network.

We suppose that there are two LSPs between the ingress-edge and the egress-edge LSRs, so the IP flow is distributed between a primary and a secondary route. The primary LSP is usually set up along the shortest path so most of the traffic should be sent through this tunnel.

Each LSR measures the transmitted bytes of its output links at constant intervals. The collected statistical information is then flooded to the network via Opaque OSPF LSAs (Link State Advertisements) extensions messages [10]. The ingress-edge LSR stores the information reported from all the LSRs. So, the ingress-edge LSR knows the traffic of each link along a LSP and can compute the load of every LSP and check whether it is congested.

The ingress-edge LSR distributes the IP flows between the primary LSP and the secondary LSP to avoid congestion of the primary path. The LSR distributes the IP flows according to a hash value [11], computed from the fields that uniquely identified an IP flow [12]. These fields are destination and source IP addresses and protocol (taken from the IP header), and destination and source ports (from the transport header).

The ingress-edge LSR splits the range of the obtained hash values between the two LSPs. For instance, the range of hash values can be 0-65535 if CRC 16 calculation is used. We call this delimitation the *hash boundary value*, figure 1.

The load balance among the two LSPs is achieved by moving the hash boundary value back and forth according to the current LSP utilization. Specifically, the load is adjusted by moving the hash boundary so that the primary LSP load should be lower than a certain congestion level. This corresponds to moving some of the IP flows carried on one LSP to the other.

Using this technique the messages of a flow usually go by the same path avoiding the traffic disorder.

3. The LCM load balance algorithm

The Load balance with Congestion and Mean utilization thresholds (LCM) algorithm was designed in order to keep the load of the primary LSP within two levels marked by the *congestion threshold* (*C*) and the *mean utilization threshold* (*M*). This algorithm was first published in [9] and after extensive checking we have introduced two changes aiming at guaranteeing that the secondary path is only used if the total load exceeds the congestion level and to reduce the total convergence time.

As explained in [9], the congestion threshold is set to the highest load value acceptable in the links (we assume a value around 60%) while the mean utilization threshold sets the return level of the algorithm (the point to send back to the primary path the traffic previously distributed to the secondary LSP) and must be fine tuned by the network administrator. As a secondary goal of the LCM algorithm the load of the primary LSP must always be higher or equal than the load of the secondary LSP.

The LCM algorithm is based on a time dependant variable called *mean load* (*L*). The mean load of a LSP is periodically computed as a function of the previously computed mean load and the current measurements, called *current_load*, taken by every LSR. Both variables are weighted by a constant value α as follows,

$$L_{[t]} = (1 - \alpha) * L_{[t-1]} + \alpha * current\_load$$

Each LSR floods the traffic load information of its output links to the network via opaque LSA messages. The ingress-edge LSR stores the information received from the LSRs involved in its LSPs and computes the *current_load* value of each LSP as the maximum load reported by any LSR of the path.

The value α is used to control the weight of the current measures versus the algorithm history. Bigger values of α produce faster responses to changes in the LSP load and traffic load balance but may introduce oscillations into the system (classical ping-pong problem). Small values of α give more weight to the past history thus avoiding oscillations but may slow the system time reaction to traffic load changes and may render the load balance almost irrelevant [7]. The network administrator is responsible for the fine tuning of this parameter.

The algorithm, shown in figure 2, works as follows:

At the beginning, the traffic load is sent via the primary LSP. When the load of the primary LSP reaches the threshold *C* the algorithm balances the traffic to the secondary LSP so that the load is shared by the two paths. Then the algorithm iterates in order to fulfil the following goals:

- To keep the load of the primary path (*Lp*) between the thresholds *M* and *C*.
- To keep *Lp* above or equal to the load of the secondary path (*Ls*), to reduce the possibility of oscillations.

- To close the secondary path as soon as the total load can be carried by the primary LSP.

It checks whether the load of the primary LSP is greater or equal than the load of the secondary LSP. If true, it tries to keep the load of the primary path within the two thresholds (*M* and *C*), moving the amount of traffic (*M-Lp*) from the secondary path to the primary path so that *Lp* reaches *M*, or moving the amount of traffic (*Lp-C*) from primary to secondary, to lower *Lp* below the congestion threshold *C*.

Otherwise the algorithm checks whether the secondary path can be closed (the total traffic load is lower than C). If true, it moves all the traffic from the secondary to the primary LSP. Else, it moves half the difference of traffic load from secondary to primary path in order to balance the load of both paths.

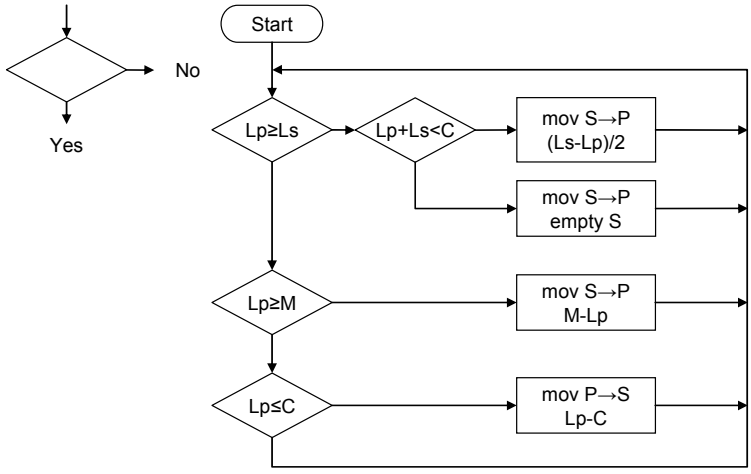

Figure 2. LCM flow diagram.

4. Test bed

In this section, we present a simple example scenario to demonstrate how the LCM algorithm works. We have deployed a MPLS network, see figure 3. There are 3 LSRs: an ingress-edge router (LSR1), an intermediate router (LSR2) and an egress-edge router (LSR3). All the LSRs are connected through 10 Mbps Ethernet links. There are two user sender processes, running on the end system 1, and the second one is running on the intermediate LSR2 and will be used to produce network congestion problems. These two processes send traffic to the receiver process running on the LSR3. As the figure 3 shows, there are 3 LSPs configured: the primary and the secondary LSPs for traffic from the sender process 1 and the LSP3 for traffic from the sender process 2.

As we said before, the primary LSP is usually set up along the shortest path, but in our test bed is set up along other path, in order to generate congestion traffic in this LSP.

The ingress-edge LSR1 will be in charge of balancing load between the primary and the secondary LSPs when the traffic from the sender process 2 overloads the link between the intermediate LSR2 and the egress-edge LSR3.

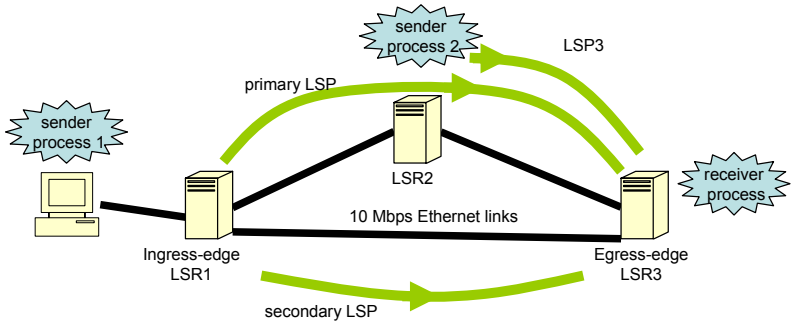

Figure 3. Network test bed.

The sender processes are running on a SuSE 7.3 Linux box. They use the *mgen* toolset [13] to generate the real-time flows. The LSRs are SuSE 7.2 Linux boxes with the MPLS stack version 1.1 for Linux [14].

In order to get an efficient load balance, the hash function must take into account the *avalanche effect* to return scattered output values from consecutive input values. Therefore, even if we only change one of the hash function input values, the result must be completely different. This behaviour gives out the traffic fairly. We have implemented the hash function proposed by Sastre [15], which is based on the Wilbur function.

We use the OSPF API to flood the link state information to the ingress-edge LSR1. The load between LSR2 and LSR3 is known by the ingress-edge LSR1 through the OSPF opaque LSA messages [16][17], which are flooded by the intermediate LSR2, according to the following procedure (more details can be found in [9]). The intermediate LSR2 calculates the bandwidth used by the primary LSP in the output interface within a period of time T (5 sec.), and then floods it using the OSPF API [18][19]. The ingress-edge LSR1 analyzes the OSPF opaque LSA packets and extracts the bandwidth value which will be used to calculate the primary LSP load (*Lp*). In order to know the loads between LSR1 and LSR2 and between LSR1 and LSR3, the LSR1 implements a procedure to obtain these local traffic statistics. Once the ingress-edge LSR1 knows the statistics of all the links of the primary and secondary LSPs, it calculates *Lp* and *Ls* and adjusts the hash boundary values. In our scenario, every time a

new opaque OSPF packet from the intermediate LSR2 arrives to the ingress-edge LSR1, the LCM algorithm is executed and *Lp, Ls* and the hash boundary are then computed.

5. Test results

Using the previous scenario (figure 3), we have carried out several experiments. C is set to 6.2 Mbps and M is set to 3 Mbps. We have configured the sender process 1 to send 100 flows of 56 Kbps each one (in total 5.6 Mbps) from the end system 1, figure 4. Once the system is stabilized we start the sender process 2 to send a different flow of 2.6 Mbps (the congestion traffic) from the intermediate LSR2.

The aggregated traffic of both senders triggers the load balance. After the mean loads are stable the test finishes so the length of each test depends on the time needed to stabilize the algorithm.

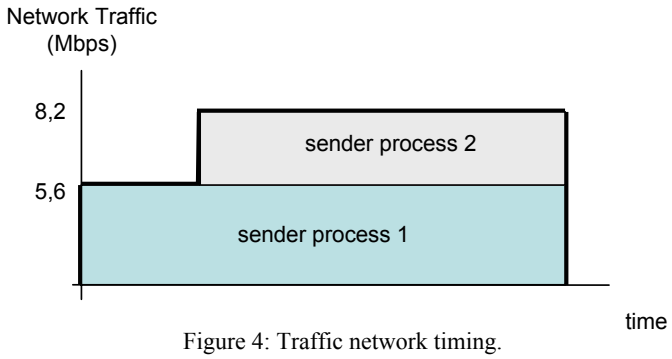

Figure 4: Traffic network timing.

We have developed a software simulator for our scenario. The simulator is written in C language and it allows us to carry out tests very fast and to validate the test bed results.

Using the previous configuration, we have tested the algorithm using some significant α values. In this paper, we present tests with three α values, a low, medium and high values (0.05, 0.5 and 1). There was no oscillation in any case. In order to better understand the behaviour of the LCM algorithm, the next figures show both the primary and secondary LSPs current load (LSR2-LSR3 and LSR1-LSR3 link traffic) and the primary and secondary LSPs mean load (*Lp* and *Ls*).

For α equal to 0.05, figures 6 and 7, the mean load changed smoothly because of the low weight of the previous load values in the load balance calculation.

The figure 5 shows the result of load balance using $\alpha = 0.05$, in the test bed. To explain the results we have split the graphic in four time periods:

- When the test starts, see time period 1, the traffic of the *Lp* (the mean load of the primary LSP) increases slowly up to 5.6 Mbps, which is lower than the congestion threshold C (6.2 Mbps), so the LCM algorithm does not balance and the links traffic remains unchanged.

- When the congestion traffic starts, see time period 2, *Lp* increases slowly and the algorithm does not balance load until *Lp* reaches the C threshold.
- During the time period 3, the LCM balances traffic from the primary to the secondary LSP. The primary LSP current load (LSR2-LSR3 link traffic) goes down. This causes *Lp* to change it and to start decreasing. Period 3 finishes when *Lp* decreases to a value lower or equal to C. Then LCM stops balancing load and the traffic links remain stable for a long time. After some time the mean load will also reach these values.
- Throughout the period 4 there is not traffic balance and *Lp* decreases to finally match the primary current load (LSR2-LSR3 link traffic).

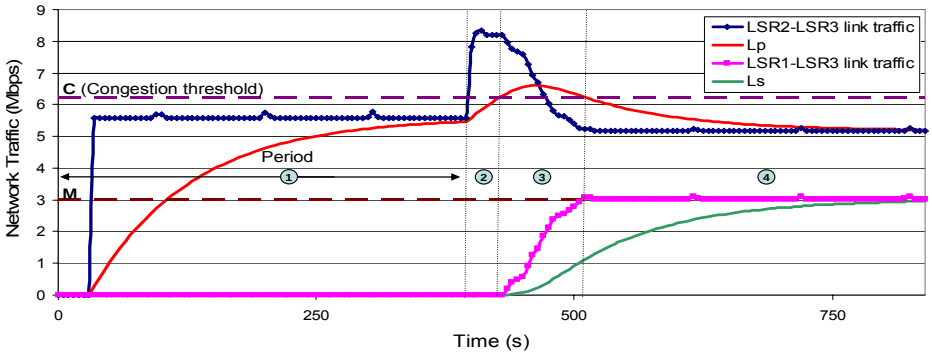

Figure 5. Load balance using $\alpha = 0.05$, test bed results.

The figure 6 shows the simulator results. It can be shown they are very close to the test bed ones.

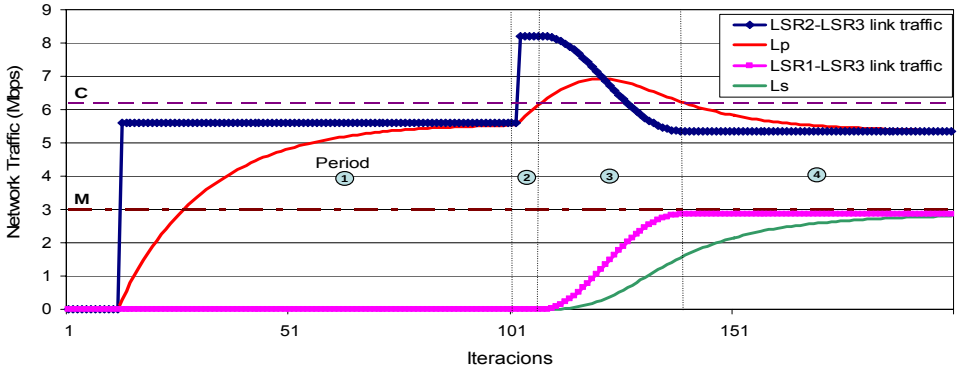

Figure 6. Load balance using $\alpha = 0.05$, simulation results.

The figures 7 and 8 show the results using $\alpha = 0.5$. The main difference with the above tests is that the mean values change more quickly so the length period of traffic balance (period 3) is shorter.

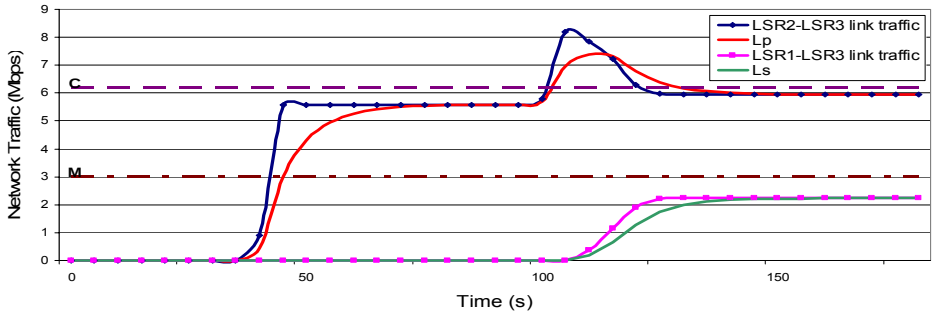

Figure 7. Load balance using α = 0.5, test bed results.

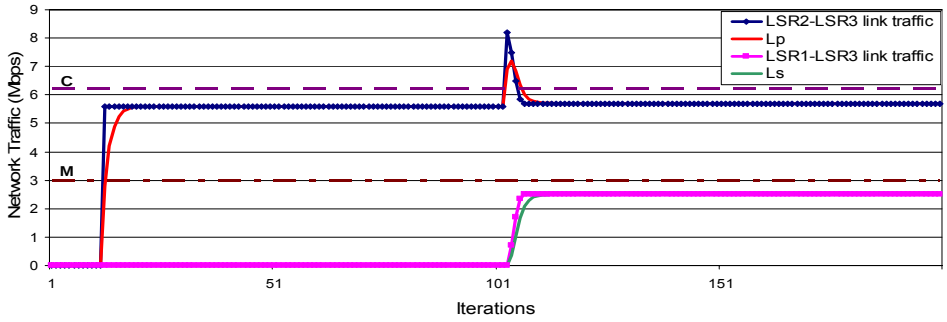

Figure 8. Load balance using $\alpha = 0.5$, simulation results.

Finally, figures 9 and 10 show the results using $\alpha = 1$. Since there is no history influence the mean load matches the current load so the balance period is the shortest possible.

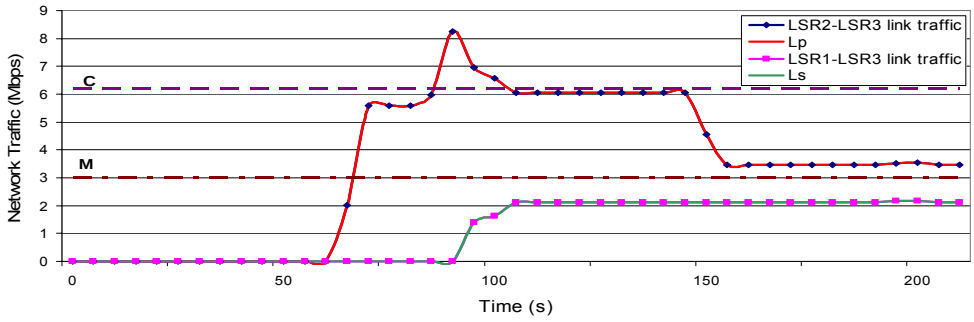

Figure 9. Load balance using $\alpha = 1$, test bed results.

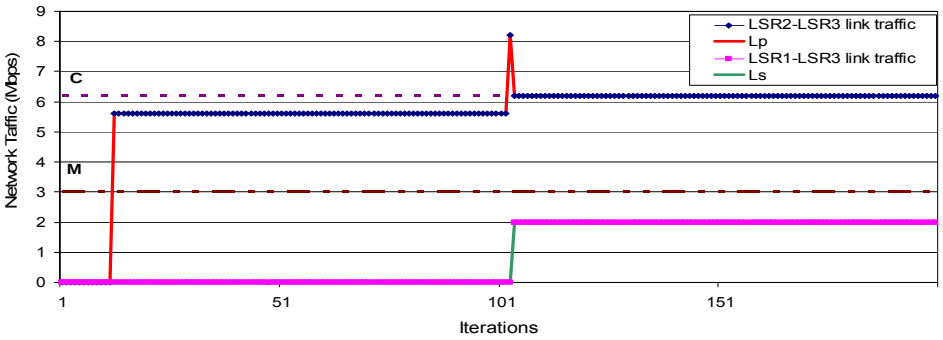

Figure 10. Load balance using $\alpha = 1$, simulation results.

The last results show the evolution of the convergence time as a function of α, figure 11. The convergence time is defined as the time elapsed from the begging of congestion traffic to the instant when *Lp* is within 1% of C, that is the time since a perturbation starts to the stabilization of the balanced traffic. The convergence time decreases with higher α values. Figure 11 compares the test bed and the simulator convergence times. In both cases the evolution of the convergence time is similar, the highest difference being 20 seconds (4 iterations).

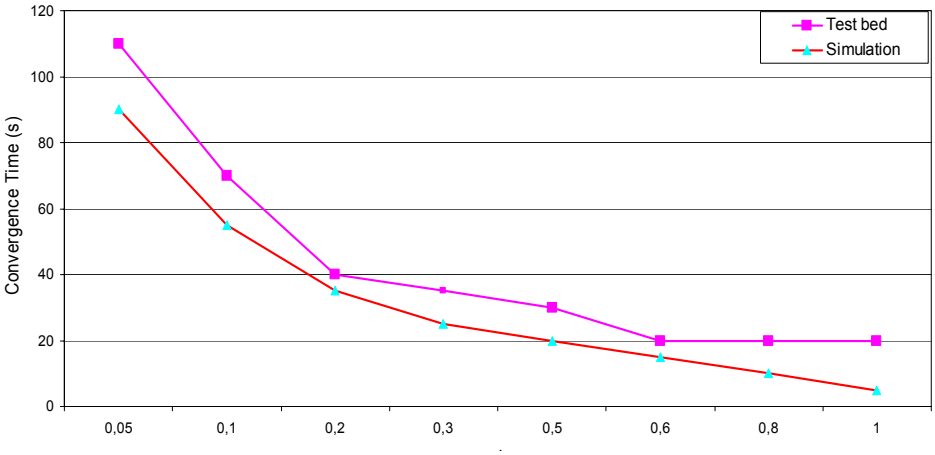

Figure 11. Convergence time.

6. Conclusions and future work

A lab test bed based on open platforms and free distribution code, has been setup to check whether the algorithm efficiently balances the load when congestion arises.

The LCM algorithm achieves the load balance in an efficient way and unlike other algorithms [7][8], shows no evidence of oscillation in the load.

A simulator of the scenario has been implemented. The results of the simulator are very close to the test bed results.

Extensive tests have been carried out to show the behaviour of the LCM algorithm with different α values.

Other experiments can be planned adding new LSPs and new traffic sources. A more complex test bed must be implemented and also the simulator must be updated.

Another line of work is to incorporate new parameters into the LCM algorithm, such as the amount of load that should be balanced, and to investigate their influence.

8. Referentes

[1] J. Moy, "OSPF Version 2". RFC2328, 1998.

[2] E. Osborne, "Traffic Engineering with MPLS". Editorial Cisco Press, julio 2002.

[3] K. Gopalan, T. Chiueh, Y. Lin, "Load Balancig routing with bandwidth-delay guarantees" IEEE Communications Magazine, June 2004.

[4] T. Ogura, M. Katoh, T. Aoyama, "Dynamic traffic engineering method with priority control" IASTED International conference, pp. 435-440. september 2003.

[5] E. Dinan, D. Awduche, B. Jabbari, "Analytical Framework for Dynamic Traffic Partitioning in MPLS Networks," IEEE International Conference on Communications (ICC-2000), New Orleans, Louisiana, June 18-22, 2000.

[6] J. Jo, , Y Kim, H. Chao, F.Merat, "Internet Traffic Load Balancing using Dynamic Hashing with Flow Volume", SPIE ITCom 2002, Boston, MA, Aug. 2002.

[7] T. Ogura J. Suzuki, A. Chugo, M. Katoh, T. Aoyama, "Stability Evaluation of a Dynamic Traffic Engineering Methodin a Large-Scale Network" IEICE Trans. COMMUN. pp 518-525, Special Issue on the Internet Technology Feb. 2003.

[8] S. Butenweg, "Two Distributed Reactive MPLS Traffic Engineering Mechanisms for Throughput Optimization in Best Effort MPLS Networks". Proceedings of the Eighth IEEE Symposium on Computers and Communications (ISCC 2003), 30 June - 3 July 2003, Kiris-Kemer, Turkey.

[9] J. M. Arco, A. García, E. Castro y J. A. Carral, "Dynamic load balance in MPLS-TE", IV Workshop in G/MPLS Networks, Gerona, Spain.

[10] R. Coltun, "RFC 2370 - The OSPF Opaque LSA Option", July 1998.

[11] Z. Cao, Z. Wang, E. Zegura, "Performance of Hashing-Based Schemes for Internet Load Balancing", pp 332-341, IEEE Infocom 2000.

[12] J. M. Arco, B. Alarcos. J.Domingo, "Programación de aplicaciones en redes de comunicaciones bajo entorno UNIX", University of Alcala, 1997.

[13] B. Adamson, "The Multi-Generator (MGEN) Toolset". http://manimac.itd.nrl.navy.mil/MGEN/

[14] J. R. Leu, R. Casellas, "MPLS for Linux" Source Forge http://sourceforge.net/projects/mpls-linux/

[15] Emilio J. Sastre García "Estudio, desarrollo y evaluación de funciones resumen utilizadas para la generación de firmas digitales". TFC de Emilio José Sastre García. Alcalá de Henares, 2004.
[16] Quagga Routing Software Suite, GPL licensed IPv4/IPv6 routing software, http://www.quagga.net/docs/docs-info.php, latest visit February 2005.
[17] Free routing software http://www.zebra.org
[18] R. Keller, "An extended Quagga/Zebra OSPF daemon supporting an API for external applications" http://www.tik.ee.ethz.ch/~keller/ospfapi/
[19] R. Keller, "Dissemination of Application-Specific Information using the OSPF Routing Protocol" Technical Report Nr. 181, TIK, Swiss Federal Institute of Technology Zurich, Switzerland, November 2003.