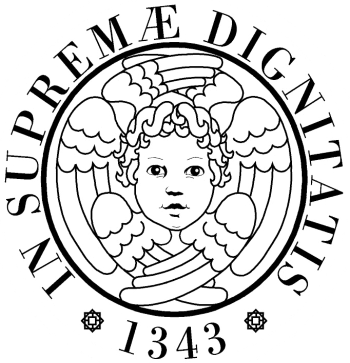
Implementazione di un tool per la configurazione di LSP MPLS-TE.

Basile Mariano, Buono Angelo

19th April 2017



Contents

1 Introduzione

1.1 LSPConfigurationTool

LSPConfigurationTool è un programma monoprocesso che permette di avere una visuale delle interconnessioni presenti in una rete e configurare percorsi Label Switched.

Le caratteristiche principali sono:

- semplicità di utilizzo
- consistenza delle informazioni topologiche anche nel caso di guasti

Tali caratteristiche conferiscono al tool usabilità, da un lato, e piena affidabilità, dall'altro, relativamente alla configurazione di TE tunnels consistenti con la reale disponibilità di risorse riservate. L'implementazione del tool, inoltre, è tale da poter garantire la sua esecuzione anche in un ambiente con più livelli di virtualizzazione presenti (come nel nostro caso).

Il tool si presenta all'utente attraverso un'interfaccia che permette di:

1. visualizzare la topologia della rete
2. visualizzare le informazioni relative ad un singolo router
3. creare LSP MPLS-TE tra coppie ingress/engress con relativo requisito di capacità
4. aggiornare esplicitamente la topologia
5. visualizzare i TE-tunnels creati
6. uscire dal tool

1.2 Overview dell'ambiente d'uso

Il tool, realizzato e testato per essere usato nell'ambiente di simulazione GSN3, è eseguito su un host che rappresenta l'istanza di una VM su cui è presente la distribuzione Linux Micro Core, anche nota come "Core", una mini-distribuzione Linux di soli 8MB.

Tale scelta è risultata necessaria poichè l'utilizzo di altre distribuzioni Linux, come la versione desktop di Ubuntu 16.04 LTS o la versione lite di Ubuntu,"lubuntu", causano inevitabilmente l'arresto forzato del simulatore GSN3 a causa dell'eccessiva quantità di risorse computazionali richieste. La motivazione deriva dal fatto che è presente un primo livello di virtualizzazione, quello in cui esegue il simulatore GNS3, ed un secondo introdotto dall'host che esegue il tool, rappresentato da un'altra istanza di VM.

Quando successivamente abbiamo constatato che il simulatore offriva già un host, tra quelli importabili, con sistema operativo Core, abbiamo avuto la conferma che tale scelta fosse la più consona. Inoltre tale host si presenta provvisto della libreria Quagga, fondamentale ai fini dell'esecuzione del tool. Attraverso il sw Quagga, infatti, è possibile ottenere le informazioni riguardo la topologia della rete.

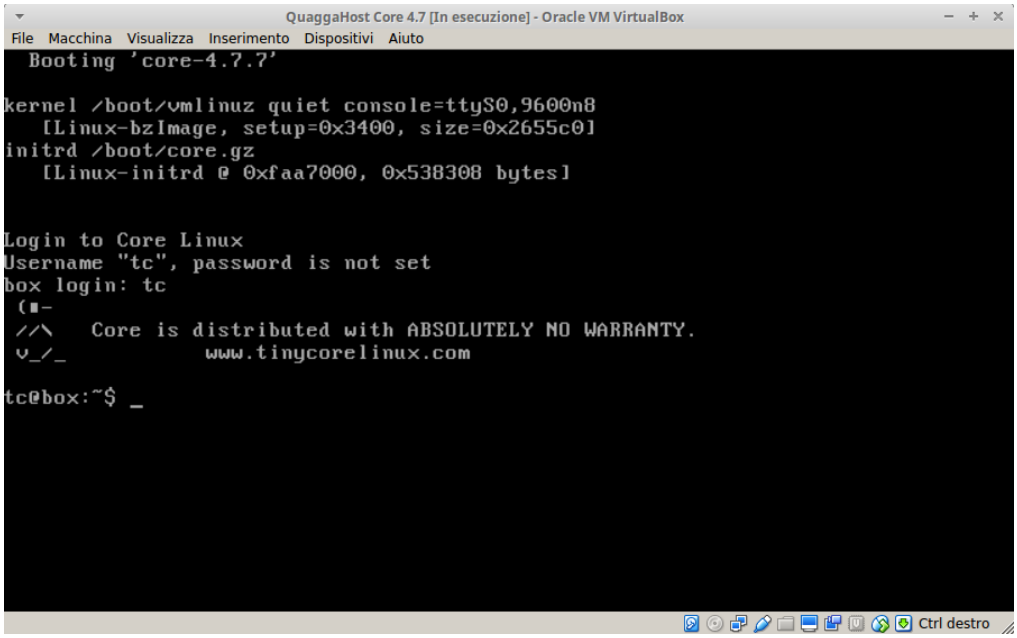


Figure 1: VM con Core 4.7.7

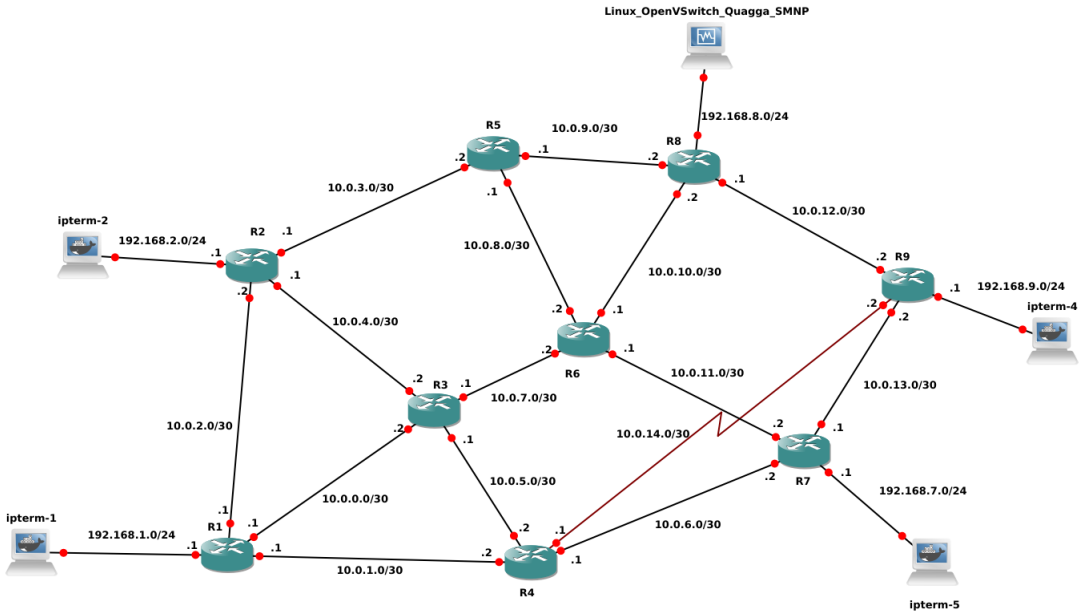
Per ovviare alla mancanza introdotta da Quagga relativa al recupero delle informazioni sul traffic engineering è risultato necessario ricorrere all'utilizzo del protocollo SNMP. Pertanto abbiamo dovuto configurare sui vari router della rete degli agent SNMP in modo tale da garantire le risposte relative alle richieste snmp effettuate dall'host su cui il tool viene eseguito. Per generare tali richieste sull'host è stato appositamente installato l'applicativo "net-snmp". Quest'ultime sono necessarie al fine di ottenere per ogni interfaccia di ciascun router le seguenti informazioni:

- capacità massima riservata per i TE tunnels
- capacità massima riservabile per singolo TE tunnel
- la capacità totale attualmente utilizzata

2 Progettazione e realizzazione del tool

2.1 La rete emulata

Per il progetto è stata elaborata e configurata appositamente una rete costituita da 9 nodi interconnessi in modo tale da garantire, per una qualsiasi coppia ingress/engress router, la presenza di più percorsi.
La rete in questione è di seguito raffigurata:



La configurazione prevede, nel dettaglio, per ogni router le seguenti caratteristiche:

- l'assegnazione di un hostname
- l'abilitazione di CEF su tutti i router della rete e su tutte le interfacce (ad eccezione di quelle collegate a degli hosts), al fine di abilitare il label switching forwarding attraverso l'implementazione proprietaria di MPLS
- l'abilitazione del protocollo DHCP per i router collegati ad una stub network
- l'abilitazione a creare tunnel traffic-engineering su tutti i router e su tutte le interfacce (eccezione fatta per quelle collegate a hosts)
- il setup delle capacità riservate per i tunnel, per mezzo del protocollo RSVP: massima capacità riservata per i tunnel e massima capacità riservabile per singolo tunnel
- la modifica alla configurazione classica di OSPF per l'abilitazione di TE-tunnels
- l'abilitazione a connessioni remote TELNET con autenticazione obbligatoria dell'utente, per poter eseguire le configurazioni dei tunnel TE (SSH non è supportato dai router cisco c3600)
- l'abilitazione di agent SNMP su ogni router

2.2 La configurazione dell' host

Ai fini dell'implementazione del tool, la distribuzione utilizzata ha richiesto l'installazione di una serie di sotware aggiuntivi, tra cui:

- "gftp", un client ftp ai fini di trasferire all'interno della macchina il codice sorgente. Core non essendo provvisto di ambiente grafico (e tantomeno della funzionalità di scrolling) risulta poco comodo da utilizzare come ambiente di sviluppo
- l'editor di testo "nano": per apportare le eventuali (piccole) modifiche al codice sorgente senza la necessità di dover utilizzare gftp
- il compilatore "gcc" e il software "compiletc" per la compilazione del codice sorgente
- la shell "bash"
- il software "expect" per la configurazione automatizzata dei TE-tunnels
- il software "net-snmp" per i motivi descritti in 1.2

2.3 Il tool

Il tool consiste di un file principale scritto in linguaggio C, del relativo header file (.h), di 4 script BASH ed 2 script EXPECT.

2.3.1 lsp.h

Il programma C fa uso di tre strutture dati: **link**, **node**, **hop**.

```
struct link{
    char srcID[16];
    char srcIntIPAddr[16];
    char destID[16];
    char destIntIPAddr[16];
    int band;
    int maxBand;
    int maxBandPerFlow;
    struct link* next;
};
/*Array of links per each router in the network*/
struct link** network;
int nrouters;
```

La struttura "link" rappresenta un link tra due router: contiene informazioni su i due router connessi e le interfacce coinvolte, con le informazioni sulla capacità dell'interfaccia "sorgente" (quella di partenza). Tali "link" sono concatenati in una lista che costituisce tutti i link di un router. Un array di "link", chiamato **network**, contiene per ogni router, identificato dall'indice dell'array, la rispettiva lista di "link", che ne rappresenta tutte le interconnessioni.

```
struct node{
    struct link* linkPointer;
    struct node* father;
    struct node* child;
    struct node* sibling;
};
```

La struttura "node" rappresenta un nodo dell'albero dei percorsi possibili da un ingress router. Ogni "node" rappresenta astrattamente un router ed una propria interfaccia. Il puntatore sibling punta ad un "node" che rappresenta un'altra interfaccia del router in esame. Il puntatore child punta al "node" successivo nel percorso. Il puntatore father punta al "node" precedente nel percorso. Il puntatore linkPointer punta al "link" che contiene le informazioni topologiche relative a tale link.

```
struct hop{
    char* interfaceIPAddr;
    struct hop* next;
};
```

La struttura "hop" è un semplice elemento di una lista di interfacce (stringhe) che costituiscono il path da configurare sul router.

2.3.2 lsp.c

L'array **network** è allocato all'avvio del tool, in seguito all'esecuzione degli script che ricavano la topologia della rete. La distruzione di tale struttura avviene ogni volta che si aggiorna la topologia della rete, per essere poi riallocato dinamicamente con le nuove informazioni. L'array è popolato, attraverso la funzione *loadTopology()*, prelevando da un file .txt le informazioni ricavate dagli script che si interfacciano con Quagga ed snmp.
L'albero dei percorsi ed il path sono allocati e distrutti ogni volta che è richiesta la creazione del tunnel. In seguito all'interazione con l'utente per ottenere le informazioni sul tunnel da allocare, viene aggiornata la topologia, e, se possibile, viene creato l'albero dei percorsi e trovato il percorso ottimo. Infine il path ottimo individuato viene automaticamente configurato sul router di ingress, attraverso uno script che vi si connette in remoto.

La funzione che implementa la creazione del percorso Label Switched è *buildLSP()*.
Tale funzione legge l'input dell'utente, in particolare ingress router, engress router e banda richiesta, attraverso *readRequiredLSP(...)*, inizializza l'albero ed il path mediante *initPathTree(...)* ed *initPath(...)* e prima di procedere alla creazione aggiorna le informazioni topologiche chiamando *refreshTopology()*.
Se i cambiamenti nella topologia non sono in conflitto con il path richiesto si procede alla creazione dell'albero attravero *buildPathTree(...)* ed alla ricerca del path migliore con *computeBestLeaf(...)*.
Qualora venisse trovato un path, l'algoritmo *createLSP(...)* compone il percorso esplicito ovvero la lista degli indirizzi ip delle interfacce su cui configurare il tunnel e *configureTunnelScript(...)* procede alla configurazione del tunnel.
Nel caso il tunnel venga configurato correttamente la funzione *updateLinkBand(...)* aggiorna localmente le capacità disponibili sulle interfacce coinvolte nel tunnel.

Bisogna spendere alcune parole sull'algoritmo che si occupa di calcolare il path migliore.

```
void computeBestLeaf(struct node* root, char* destID, int bandw, float availUtilization,
    int pathLen, struct node** bestLeaf, float *bestAvgAvailUtilization, int *bestLen){

    float actualAvailUtilization,temp,actualAvgAvailUtilization;

    temp=((float)(root->linkPointer->band-bandw))/((float)root->linkPointer->maxBand);
```

```
    actualAvailUtilization = temp + availUtilization;

    /*Found a Valid Path (Leaf reached is okay wrt band req and destination!)*/
    if(strcmp(root->linkPointer->destID, destID) == 0 &&
        root->linkPointer->band >= bandw &&
        root->linkPointer->maxBandPerFlow >= bandw){

        actualAvgAvailUtilization = actualAvailUtilization/(pathLen+1);

        /*Found a Better Path (Leaf reached is better than the actual best Leaf)*/
        if( (*bestAvgAvailUtilization == -1 && *bestLen == -1) ||
            (actualAvgAvailUtilization - *bestAvgAvailUtilization > PERC_AVAIL_GAP) ||
            ((fabs(actualAvgAvailUtilization - *bestAvgAvailUtilization) <=
                PERC_AVAIL_GAP) && (pathLen + 1 < *bestLen)) ){

                *bestLeaf = root;
                *bestAvgAvailUtilization = actualAvgAvailUtilization;
                *bestLen = pathLen + 1;
        }
    }
    /*Descend the Tree on the child*/
    else if(root->linkPointer->band >= bandw &&
            root->linkPointer->maxBandPerFlow >= bandw && root->child != NULL)

        computeBestLeaf(root->child, destID, bandw, actualAvailUtilization, pathLen + 1,
            bestLeaf, bestAvgAvailUtilization, bestLen);


    /*Explore if siblings (another link on the current router)*/
    if(root->sibling != NULL)

        computeBestLeaf( root->sibling, destID, bandw, availUtilization, pathLen,
            bestLeaf, bestAvgAvailUtilization, bestLen );
}
```

Il criterio di selezione del path ottimo confronta i vari percorsi in base al valore della funzione obiettivo: $max(\frac{1}{n} * \sum_{i=1}^{n} \frac{residualBand_i - requestBand}{reservedBand_i})$.
Viene considerata la media delle capacità residue percentuali (rapporto tra capacità disponibile dopo allocazione e capacità massima riservata) dei link del percorso, che si avrebbe in seguito all'eventuale allocazione del path. La differenza al numeratore tiene conto delle possibili saturazioni dei link del percorso. L'algoritmo prevede, qualora le funzioni obiettivo dei path in esame differiscano per meno della soglia "PERC_AVAIL_GAP", la scelta del path ottimo basandosi sul minor numero di hop. Il valore di tale soglia, considerate le capacità da noi riservate (da qualche Mbps a qualche decina di Mbps), è stato settato a 0.01, ovvero l'1%. Pertanto si ritengono significative le differenze che vanno da qualche Kbps a qualche centinaio di Kbps. Naturalmente questa soglia può essere aggiustata ad hoc a seconda delle effettive esigenze.

2.3.3 Complessità di storage e occupazione di memoria

Il tool prevede la costruzione dell'array **network** ogni volta che la topologia della rete viene aggiornata; ciò avviene automaticamente ogni volta che viene richiesto un tunnel. Inoltre, in tale circostanza, viene anche costruito l'albero dei percorsi possibili ed eventualmente la lista degli hop costituenti il path. La memoria totale allocata e la complessità di storage richiesta in queste operazioni è:

1. Aggiornamento della topologia della rete (costruzione dell'array network):
    - Memoria allocata:

n_routers * n_interf_per_router * sizeof(struct link)
sizeof(struct link) = 80Byte

- Complessità di storage:

  ≈O(n_routers)

2. Costruzione dell'albero dei percorsi ammissibili:

   - Memoria allocata:

     n_path_possibili * sizeof(struct node)
     sizeof(struct node) = 16Byte

   - Complessità di storage:

     ≈O(n_routers^2)

3. Costruzione della lista degli hop del percorso:

   - Memoria allocata:

     lunghezza_path * sizeof(struct hop)
     sizeof(struct hop) = 8Byte

   - Complessità di storage:

     ≈O(n_routers)

L'allocazione di tali strutture dati avviene dinamicamente e la memoria viene liberata non appena possibile; fatta eccezione per la fase di costruzione del path e della configurazione, l'unica struttura dati presente in memoria è l'array **network**. L'occupazione di memoria sarà pertanto molto bassa (i.e. al massimo 200.8KB in caso di 100 router con 5 interfacce).

2.4 Gli script

2.4.1 quagga_config.sh

Tale script è invocato dal programma principale, "lsp.c", solo ed esclusivamente all'avvio del tool. Lo script provvede essenzialmente a:

- configurare l'interfaccia ethernet, eth0, tramite la quale l'host è connesso alla rete. La configurazione di tale interfaccia (indirizzo ip e maschera di rete) è acquisita in input dall'utente e validata
- abilitare il protocollo di routing ospf al fine di ricostruire la topologia della rete
- salvare tali informazioni nei file di configurazione

```
vtysh
  -c 'configure terminal'
  -c 'interface eth0'
  -c 'ip address '$ip_address'/'$netmask''
  -c 'link-detect'
  -c 'exit'
  -c 'router ospf'
  -c 'network '${ip_addr[0]}'.'${ip_addr[1]}'.'${ip_addr[2]}'.0/'$netmask' area 1'
vtysh
  -c 'write'
```

Ad una seconda esecuzione del tool, al fine di evitare inconsistenza nella configurazione dell'host (utente che inserisce indirizzo ip e maschera diversi dai precedenti), è necessario cancellare la configurazione precedente e riconfigurare nuovamente l'interfaccia eth0.

```
vtysh
  -c 'configure terminal'
  -c 'interface eth0'
  -c 'no ip address '$eth0IpAddress'/'$eth0Netmask''
  -c 'exit'
  -c 'router ospf'
  -c 'no network '${eth0IpAddressOctet[0]}'.'${eth0IpAddressOctet[1]}'.
      '${eth0IpAddressOctet[2]}'.'0'/ '$eth0Netmask' area 1'
vtysh
  -c 'write'
```

Bisogna notare che dal momento dell'attivazione di ospf al momento della creazione del relativo database trascorre una quantità di tempo variabile. Tale quantità dipende da vari fattori: principalmente il numero di router attivi nella rete e l'instante di tempo in cui si attiva il router a cui l'host è connesso. Per avere una situazione in cui il processo ospf è a regime, cioè l'intera topologia è ricostruibile, si è deciso, dopo diverse prove, di settare tale quantità di tempo a 20 secondi.

```
startQuagga 20
```

2.4.2 get_topology.sh

Tale script viene invocato in tre occasioni:

- all'avvio del tool dallo script "quagga_config.sh"
- dal programma principale "lsp.c" nel caso di esplicita richiesta di aggiornamento delle informazioni locali
- dal programma principale "lsp.c" nel caso di richiesta di configurazione di un nuovo TE-tunnel

Lo script provvede essenzialmente a:

- ricostruire la topologia della rete.
- individuare le capacità residue disponibili sui vari collegamenti.

La ricostruzione topologica avviene, nella sua completezza, attraverso il database generato da protocollo ospf. In prima istanza si provvede a recuperare i vari router id associati ai router presenti nella rete.

```
ip_address = $(cat /usr/local/etc/quagga/zebra.conf | grep "ip address" | cut -d " " -f4 | cut -d / -f1)

first_line_network_state_section = $(vtysh -c "show ip ospf database" |
                                     grep -nr "Net Link States" | cut -d : -f 1)
if [ "$first_line_network_state_section" == "" ];
then
echo -e "\n========== ROUTER R8 HAS TO BE SWITCHED ON ==========!!\n"
exit 1
fi

last_line_router_state_section = $((first_line_network_state_section-2))

first_line_router_state_section = $((last_line_router_state_section-6))

router_state_section = $(vtysh -c "show ip ospf database" | head -$last_line_router_state_section |
                                     tail -$first_line_router_state_section)

routerId = ( $(echo "$router_state_section" | awk '{print $1;}') )
```

Qualora anche solo uno dei router già presenti nel database, a causa di eventuali guasti hardware o software, interrompesse l'invio degli annunci OSPF di tipo 1, si continuerebbe ad avere nel database OSPF una entry a lui riservata. Per tale motivo, prima di ricostruire la topologia, si anticipa l'invio delle richieste snmp, per il recupero dei tre valori di capacità, in modo da individuare tali router inattivi. In questo modo è possibile procedere con la ricostruzione della topologia considerando esclusivamente i router effettivamente attivi.

```
rsvpIfBPerFlow = ( $(snmpbulkwalk -v2c -c $community ${routerId[$i]} 1.3.6.1.2.1.52.1.1.1.3 |
        awk -F ": " '{print $2;}') )

if [ "${#rsvpIfBPerFlow[@]}" -eq 0 ]
then echo -en "Router ID: ${routerId[$i]} "
continue
fi

rsvpIfB = ( $(snmpbulkwalk -v2c -c $community ${routerId[$i]} 1.3.6.1.2.1.52.1.1.1.2 |
        awk -F ": " '{print $2;}') )

if [ "${#rsvpIfB[@]}" -eq 0 ]
then echo -en "Router ID: ${routerId[$i]} "
continue
fi

totalUsedRsvpB = ( $(snmpbulkwalk -v2c -c $community ${routerId[$i]} 1.3.6.1.2.1.52.1.1.1.1 |
        awk -F ": " '{print $2;}') )

if [ "${#totalUsedRsvpB[@]}" -eq 0 ]
then echo -en "Router ID: ${routerId[$i]} "
continue
fi

activeRouter[$acIndex] = ${routerId[$i]}
acIndex=$((acIndex+1))
```

A questo punto per ciascuno dei router attivi si procede, in prima istanza, all'individuazione dei rispettivi router collegati per mezzo di collegamenti seriali, se presenti.

```
for (( i=0; i<${#activeRouter[@]}; i++ ))
do
    neighborRouterId=( $(vtysh -c 'show ip ospf database router '${activeRouter[$i]}'' |
        grep "Neighboring Router ID:" | cut -d : -f 2) )

    k=0
    if [ ${#neighborRouterId[@]} -gt 0 ];
    then
        counterNeighborn[$i] = ${#neighborRouterId[@]}
        p2p[$i]=1
        for (( j=0; j<${#neighborRouterId[@]}; j++ ))
        do
            p2pRouterId[$i,$j] = ${neighborRouterId[$j]}

            neighbornRouterIdLineNumber = $(vtysh -c 'show ip ospf database router '${activeRouter[$i]}'' |
                grep -n "Neighboring Router ID: ${neighborRouterId[$i]}" |cut -d : -f 1)

            neighbornRouterIdLineNumber = $((neighbornRouterIdLineNumber+1))

            localIpAddrIf[$i,$j] = $(vtysh -c 'show ip ospf database router '${activeRouter[$i]}'' |
                head -n $neighbornRouterIdLineNumber | tail -1 | cut -d : -f 2 | cut -d " " -f 2)

            targetIpAddr[$k] = ${localIpAddrIf[$i,$j]}
            k=$((k+1))

            neighbornRouterIdLineNumber =
                $(vtysh -c 'show ip ospf database router '${neighborRouterId[$j]}''|
                grep -n "Neighboring Router ID: ${activeRouter[$i]}" | cut -d : -f 1)

            neighbornRouterIdLineNumber = $((neighbornRouterIdLineNumber+1))

        p2pRouterIpAddrIf[$i,$j] = $(vtysh -c 'show ip ospf database router '${neighborRouterId[$j]}'' |
            head -n $neighbornRouterIdLineNumber | tail -1 | cut -d : -f 2 | cut -d " " -f 2)
        done
    fi
```

Si procede poi con l'individuazione dei rispettivi router collegati per mezzo di collegamenti fastEthernet.

```
designatedRouterIpAddress = ( $(vtysh -c 'show ip ospf database router '${activeRouter[$i]}'' |
    grep "Designated Router address:" | cut -d : -f 2) )

routerIfAddress = ( $(vtysh -c 'show ip ospf database router '${activeRouter[$i]}'' |
    grep "Router Interface address:" | cut -d : -f 2) )

for (( j=0; j<${#designatedRouterIpAddress[@]}; j++ ))
do
    if [ "${designatedRouterIpAddress[$j]}" != "${routerIfAddress[$j]}" ];
    then
        ipAddrIf[$i,${counter[$i]}] = ${routerIfAddress[$j]}

        attachedRouterId[$i,${counter[$i]}] =
            $(vtysh -c 'show ip ospf database network '${designatedRouterIpAddress[$j]}'' |
            grep "Advertising Router:" | cut -d : -f 2 | cut -d " " -f 2)

        attachedRouterIpAddrIf[$i,${counter[$i]}] = ${designatedRouterIpAddress[$i,$j]}

        for (( w=0; w<${#activeRouter[@]}; w++ ))
        do
            if [ "${activeRouter[$w]}" == "${attachedRouterId[$i,${counter[$i]}]}" ];
            then
                attachedRouterId[$w,${counter[$w]}] = ${activeRouter[$i]}

                attachedRouterIpAddrIf[$w,${counter[$w]}] = ${routerIfAddress[$j]}

                ipAddrIf[$w,${counter[$w]}] = ${attachedRouterIpAddrIf[$i,${counter[$i]}]}

                counter[$i] = $((counter[$i]+1))

                counter[$w] = $((counter[$w]+1))
            break
            fi
        done
    fi
done
```

A questo punto l'intera topologia e i tre valori di capacità richiesti, per ciascuna interfaccia di ciascun router, possono essere salvati su file, per essere poi letti dal programma principale "lsp.c".

```
echo "$acIndex" >> topology.txt
for (( i=0; i<${#activeRouter[@]}; i++ ))
do
    for (( j=0; j<${counter[$i]}; j++ ))
    do
        echo "${activeRouter[$i]} ${ipAddrIf[$i,$j]} ${attachedRouterId[$i,$j]} ${attachedRouterIpAddrIf[$i
            ,$j]} ${rsvIfB[${ipAddrIf[$i,$j]}]} ${rsvIfBPerFlow[${ipAddrIf[$i,$j]}]} $((rsvIfB[${ipAddrIf[
            $i,$j]}]-totalUsedRsvB[${ipAddrIf[$i,$j]}]))" >> topology.txt
    done
    if [ "${p2p[$i]}" -ne 0 ];
    then
        for (( w=0; w<${#counterNeighborn[$i]}; w++ ));
        do
            echo "${activeRouter[$i]} ${localIpAddrIf[$i,$w]} ${p2pRouterId[$i,$w]} ${p2pRouterIpAddrIf[$i,
                $w]} ${rsvIfB[${localIpAddrIf[$i,$w]}]} ${rsvIfBPerFlow[${localIpAddrIf[$i,$w]}]} $((
                rsvIfB[${localIpAddrIf[$i,$w]}]-totalUsedRsvB[${localIpAddrIf[$i,$w]}]))" >> topology.txt
        done
    fi
done
```

2.4.3 configure_tunnel.sh

Tale script è invocato dal programma principale, "lsp.c", per mezzo della syscall "system()" in seguito alla richiesta di creazione di un TE-tunnel. La chiamata a tale script avviene solo ed esclusivamente in caso di presenza di un percorso dal router ingress al router engress che soddisfa il requisito di capacità specificato. Lo script provvede essenzialmente ad:

- acquisire e validare username e password, per la configurazione in remoto (telnet) del TE-tunnel
- acquisire e validare nome del tunnel e nome del path esplicito da configurare

Lo script, poi, invoca lo script "configure_tunnel.exp" con le informazione necessarie per la configurazione del TE-tunnel:

- le credenziali per l'autenticazione telnet
- i router id di ingress e engress router
- la capacità da allocare per il tunnel
- i "next-router" del path esplicito
- nome del tunnel e nome del path esplicito

```
./configure_tunnel.exp $username $password $1 $2 $3 $4 $tunnelName $LSP
exit $?
```

Lo script "configure_tunnel.sh" termina la sua esecuzione con una *exit $?*. Lo statement *$?* rappresenta il risultato dell'ultima istruzione eseguita, ovvero la chiamata a "configure_tunnel.exp". Tale script esegue, a sua volta, come ultima istruzione una *exit value* dove value rappresenta un intero positivo. Qualora un tunnel venisse correttamente configurato *value* avrà valore 0, altrimenti 1.
Lo statement *$?* assumerà pertanto solo questi due possibili valori. Il programma chiamante, "lsp.c", di conseguenza potrà esaminare il valore di ritorno della primitiva *system()* e solo se tale valore risulta essere uguale a 0 procede ad aggiornare le capacità residue sui router del path individuato.

2.4.4 configure_tunnel.exp

Tale script fa uso del linguaggio di scripting *Expect* che opera nella stessa maniera con cui un utente interagisce con un sistema di elaborazione: l'utente digita un comando e si aspetta una risposta dal sistema. Quando viene ricevuta la risposta a un comando l'utente può inserire un nuovo comando, e cosi via. Expect funziona esattamente nella stessa maniera ad eccezione del fatto che l'utente deve fornire allo script sia i comandi che la risposte attese.
Nel nostro scenario quindi, i comandi e le risposte attese sono esattamente gli stessi che l'utente avrebbe fornito all'ingress router se fisicamente li presente.

```
expect "Username:"
send -- "$username\r"
expect "Password:"
send -- "$password\r"
expect "#"
send -- "conf t\r"
expect "#"
send -- "interface $tunnelName\r"
expect "#"
send -- "ip unnumbered loopback0\r"
expect "#"
send -- "tunnel destination $engressRouter\r"
expect "#"
send -- "tunnel mode mpls traffic-eng\r"
expect "#"
send -- "tunnel mpls traffic-eng autoroute announce\r"
expect "#"
send -- "tunnel mpls traffic-eng bandwidth $requiredBand\r"
expect "#"
send -- "tunnel mpls traffic-eng path-option 1 explicit name $pathName\r"
expect "#"
send -- "exit\r"
expect "#"
send -- "ip explicit-path name $pathName enable\r"
expect "#"
```

```
foreach ip_addr $lsp {
send -- "next-address $ip_addr\r"
expect "#"
}
send -- "exit\r"
expect "#"
send -- "exit\r"
expect eof
exit 0
```

Dal momento in cui l'utente richiede la configurazione di un tunnel-TE al momento in cui il router di ingress viene contattato ci potrebbero essere cambiamenti topologici della rete:

- Il router ingress appare non raggiungibile da remoto, impedendo di conseguenza la configurazione del tunnel:

In tale circostanza, allo scadere di un timeout fissato a 10 secondi (per default), un messaggio avviserà l'utente dell'accaduto e il controllo ritornerà al programma principale, "lsp.c".
In questo caso, leggendo un valore di ritorno pari a 1, non verranno aggiornate le capacità residue sui link del percorso, così da mantenere la consistenza delle informazioni salvate, e sarà mostrato un messaggio che avviserà dell'impossibilità di creare il tunnel.

- Un router dell'LSP appare non raggiungibile, generando una situazione di temporanea inconsistenza:

In tale circostanza, pur essendo la configurazione sul router di ingress una configurazione valida, il protcollo rsvp non sarebbe in grado di garantire l'allocazione della capacità richiesta sull'interfaccia di tale router. Tuttavia, localmente al tool, le capacità sarebbero aggiornate anche per quella interfaccia di tale router. Si presenterbbe quindi una inconsistenza temporanea. Quest'ultima verrebbe comunque risolta non appena viene richiesto un eplicito aggiornamento delle informazioni locali (topologia e capacità residue) o la creazione di un nuovo TE-tunnel. In questi casi, infatti, le informazioni locali vengono aggiornate e le eventuali incosistenze presenti cessano di esistere.

```
expect {
    "telnet: can't connect to remote host ($ingressRouter): Network is unreachable" {
    puts "\nCONNNECTION TO $ingressRouter FAILED! Exiting..\n"
    sleep 2
    exit 1
}
```

2.4.5 show_tunnel.sh

Tale script è invocato dal programma princiaple, "lsp.c", per mezzo della syscall "system()" in seguito a richiesta esplicita dell'utente per mezzo del comando *5* del tool. Lo script provvede essenzialmente ad:

- acquisire e validare username e password, per la connessione in remoto (telnet) al router ingress

- acquisire e validare il routerID dell'ingress router

Se il routerID specificato è un routerID valido lo script procede con l'invocazione di "show_tunnel.exp" passandoli come parametri le sole credenziali necessarie per l'autenticazione telnet e il routerID del quale si vogliono visualizzare le informazioni relative agli LSPs creati.

```
./show_tunnel.exp $username $password $1
```

2.4.6 show_tunnel.exp

Lo script procede semplicemente con l'invio del comando "*show mpls traffic-eng tunnels*".

```
expect "Username:"
send -- "$username\r"
expect "Password:"
send -- "$password\r"
expect "#"
send -- "show mpls traffic-eng tunnels\r"
expect {
        -ex "--More--" {
                send -- "\r"
                set end 0
                while {$end == 0} {
                        expect {
                                -ex "--More--" {
                                        set buttonPressed [gets stdin]
                                        send -- "\r"
                                }
                                "#" {
                                        set end 1
                                }
                        }
                }
                send -- "\r"
                send -- "exit\r"
                expect "#"
                send -- "exit\r"
                expect eof
                exit 0
        }
}
```