Demystifying MPLS

The MPLS framework in OpenBSD

by Claudio Jeker

Work on supporting MPLS started in 2008 at the n2k8 mini-hackathon in Ito (Japan). In the last 2 years much work went into this new framework. Apart from the network stack changes ldpd(8) -- the label distribution protocol daemon -- was developed and bgpd(8) was modified to make it possible to setup and terminate MPLS VPNs on OpenBSD. OpenBSD is probably the first open-source system able to do MPLS out of the box without additional patches.
Most people have heard about MPLS but how it actually works is often unknown. MPLS changes the way networking is done. While the label switching part itself is trivial it is just one part of a much larger puzzle. There are changes in many routing protocols and with over 150 RFC about MPLS shows that this is more than just simple label switching.

Overview

In the late 1990s the Internet boom started and with this boom the routing table started to grow exponentaly. When router needed to store more then 100'000 routes for the full view problems started to show up. It was hard to upgrade the memory on many systems and the lookup times on large routing tables grew so much that it seemed almost impossible to do line rate best prefix matches. IP address lookups are more complex because both the address and the netmask need to be considered and the best match must be found. On the other hand perfect matches like the MAC address lookups done by switches are much simpler since there is no netmask to consider and such lookups can be done in hardware with CAM (content addressable memory) tables or with simple hash tables or binary trees. So a solution was created that allows less complex, faster and hopefully cheaper devices in the core. The IP route lookup was moved to the systems at the border of the network and inside simple label switching was performed. Other networks like Frame Relay and ATM already used label switching so the concept was not new. Unlike ATM the label switching in MPLS is done in a less complex way plus the network layer was not changed. The complexity of ATM and the small data cells are probably the cause that MPLS was created instead of switching the infrastructure over to ATM in the core. ATM is just not competitive against Ethernet when large amount of bulk data has to be transferred.

Label Switching

By default on every hop an IP route lookup has to be done. So every router along the path to the destination does his own lookup and decision. Label switching changes this so that only the edge routers of an MPLS network are doing a route lookup. The edge systems (PE router) decide which path a packet needs to take. So label switching is more like source routing since the route lookup is done on the source and then no other IP lookup needs to be done until the end of the path is reached. In the OSI model MPLS is somewhere between the data link layer (layer 2) and the network layer (layer 3). It can be viewed as an extension of the data link layer but this is not entirely correct so it is common to talk about layer 2.5 for MPLS. Each Label Switching Path (LSP) has a label assigned. The labels are not globally defined, they're swapped on every hop through the network. So the Label Switching Routers (LSR) inside the network (P

routers) don't need to look at the IP header anymore, they use the MPLS label to lookup the LSP and swap the label with the new label for the next hop. For Ethernet it is common to assign the labels per system so it does not matter on which interface a packet is received on. In ATM and Frame Relay networks labels are normally assigned per interface since the protocol labels itself are used for MPLS. OpenBSD currently only supports a system wide label space since neither Frame Relay nor ATM is supported and so per interface label spaces are not needed.

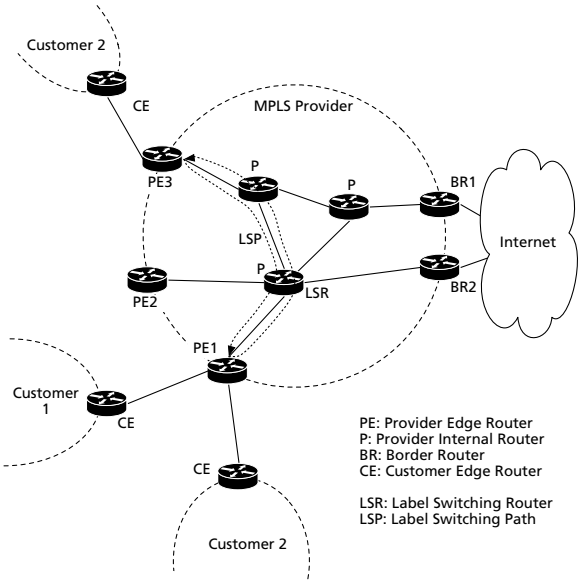

Figure 1: Overview of an MPLS Network

An LSP is an unidirectional path so it is impossible to know from where a packet was received. If an LSR along the path has troubles with forwarding the packet no error message can be sent back to the source. This makes debugging issues in label switching networks a challenge. A path needs to be manually followed hop by hop until the failure is found.

MPLS

The MPLS header as defined by RFC3032 is a simple 4-byte header consisting of a label value, a traffic class field formerly known as experimental bits, a bottom of stack flag, and a TTL.

MPLS header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ Label
|                Label                  | Exp |S|      TTL      | Stack
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ Entry

                Label:  Label Value, 20 bits
                Exp:    Experimental Use, 3 bits
                S:      Bottom of Stack, 1 bit
                TTL:    Time to Live, 8 bits
```

It is possible to stack MPLS headers and by doing that it is possible to build virtual circuits between Provider Edge (PE) routers. The last MPLS label has the bottom of stack flag set.
The TTL field can be inherited from the IP packet and on exiting the MPLS network the TTL value can be copied back to the IP packet. This plus a bit of trickery in the MPLS error handling allows to traceroute through MPLS networks. If the TTL is not copied back then the MPLS network will appear as a single hop in traceroutes.
There is no next protocol identifier so the protocol of the packet that is MPLS encapsulated needs to be known by the two ends of the LSP. Intermediate systems can only guess the next protocol by looking at the first nibble and decide if it is IPv4, IPv6 or something else. This guessing is enhanced by the suggested encoding rules in RFC4928.
Every label specifies a specific path. Because of traffic engineering it is possible to have multiple pathes to the same endpoint. A common term for such an endpoint is the forwarding equivalence class (FEC). All packets to the same FEC are treated the same way, they take the same path and get the same forwarding treatment.

Label Distribution and Forwarding

While it is theoretically possible to set up the LSPs all manually it is not feasible on non trivial networks. An automated way to assign and distribute labels is needed. For MPLS the label distribution protocol (LDP) is used for this task. By default LDP reads the routing table and assigns a label for every prefix in the table. So by default each prefix is considered an FEC and gets an own LSP. These labels are then redistributed by LDP to the other directly connected routers in the network. So the full LSP is built hop by hop from the destination to the various sources. Assigning labels is fairly trivial since they only need to be unique per system. Every LDP instance along an LSP chooses its own labels and this can be done even before the mapping for the LSP nexthop reached the router. Since LDP only reads the routing table and assigns labels according to this table it is obvious that an other routing protocol is needed to build this routing table in the first place. This is the job of a traditional IGP (internal gateway protocol) like OSPF or RIP. So while for example OSPF is used to distribute all routes through the network and there for making all routers reachable, LDP is used to distribute the LSP or label information for these routes (FEC). MPLS will only be used if both the route and label for this route are available at a router. The received label is attached to the nexthop information of the route and is from now on used for outgoing packets to that destination. A second table maps the local incoming

label for this FEC to the nexthop and outgoing label of the route. PE routers at the edge use the first table to figure out the LSP to use to get to the destination while P routers only look at the second table to switch the incoming MPLS packets forward. The label of the incoming MPLS packet is used to lookup the instructions to forward the packets. By default the lookup will return a new outgoing label that is swapped with the incoming one plus a nexthop to where the packet needs to be sent to. It is also possible that the label just needs to be popped of the stack or that a label is pushed onto the label stack. Normally only the first label on the stack is looked up. In some cases it can be possible that a pop operation causes a second lookup. P systems should only swap labels whereas PE systems initially push one or more labels onto the label stack and then pop them off at the other end.

MPLS VPN

PE routers build the initial MPLS label stack. By building a label stack with multiple labels it is possible to select an endpoint plus an additional label that the endpoint can use to select the correct customer to forward the packet to. So while packets use the same LSP to get from PE-1 to PE-2 they do not interact with either the underlying network or with other customers. So each customer is setup in a virtual routing instance - the term Virtual Routing & Forwarding (VRF) is commonly used for such instances. On OpenBSD this can be achieved with the routing domains (rdomain) support.

It is obvious that the PE routers need to exchange the labels used for the MPLS VPNs so that the correct labels can be selected for each customer. There are two ways to do this. First of all it is possible to use LDP to distribute these labels across a targeted session between two PEs. Now targeted session need to be setup manually and on large networks a large mesh with many sessions is needed and it is hard to do extensive filtering in LDP. Because of these limitations it became common to use BGP to distribute the MPLS VPNs. BGP can be easily extended by additional TLV attributes, has extensive filtering capabilities and thanks to route-reflectors it is possible to avoid full meshes between all PE routers. BGP MPLS VPNs use an own address encoding and BGP extended communities to transport all needed informations between the various PE routers.

PE - CE Interaction

For simple customer networks the interaction between the customer and the MPLS provider is fairly limited. On the customer edge (CE) router a few static routes are installed and the same is done on the PE routers. If the number of customer networks grow or it is necessary to allow dynamic routing between the sites all becomes a bit more complex. Having a routing protocol running between the CE and PE systems would allow changes to the customer network without needing changes done on the provider systems. At the same time the provider does not want that the customer could cause any problems on the provider equipment that would influence other customers. Now since the PE routers have a VRF instance assigned to each customer it is possible to run a routing protocol inside these instances. In most cases OSPF or RIP is used to exchange the routes.

So the CE router redistributes its local networks to the PE router. The PE router will then import these routes into BGP MPLS VPN and transport the networks to the other PEs that have VRFs for the customer configured. On those PEs BGP will import the routes back into their VRF instance and the local OSPF will redistribute the routes to the local CE systems. This sounds all nice but has some drawbacks. All IGP specific routing information like metrics are lost between the PE routers and it looks like the PE is the source of the networks announced to the CE routers even though these networks were announced by another customer system. This does not matter if the MPLS VPN is the only connection between the various customer networks. If this is not the case ugly things may happen. Internal routes may no longer be preferred and traffic may suddenly loop through the MPLS VPN that does not need to leave the local network. To fix this OSPF and BGP were extended so that BGP is able to transmit the full link state DB of OSPF to the other systems without losing information. Additionally OSPF was extended to flag routes inserted by PE systems to prevent loops and silly routing issues. These extensions are currently not available in OpenBSD. The interaction between bgpd and multiple ospfds is a bit tricky since the information needs to be queried over the IMSG socket.

VPLS and PWE3

MPLS VPN are used to connect customers at Layer 3 but most often customers would like a Layer 2 service. This is where Pseudowire Emulation Edge-to-Edge (PWE3) comes into play. PWE3 defines the encoding of Layer 2 packets and streams. Not only Ethernet frames can be packed into PWE3 packets but also ATM, Frame Relay, PPP, HDLC and various synchronous data stream protocols like TDM and STM can be moved over an MPLS backbone. PWE3 is a point to point protocol - for ethernet this would correspond to a simple transparent bridge. If multiple sites need to be connected via a single Layer 2 cloud it is possible to span multiple PWE3 tunnels and connect the various tunnels to a central switch or use VPLS. Virtual Private LAN Services (VPLS) use the same

Ethernet encoding as PWE3 but use a virtual switch to connect all sites together and there for supports multicast and broadcast to all sites. The setup of the VPLS and PWE3 FEC is done via LDP or BGP (PWE3 only supports LDP). Neither VPLS nor PWE3 is currently supported in OpenBSD but will come in the near future.

MPLS and OpenBSD

The OpenBSD MPLS network stack is based on the work by the Ayame project[7] but got massively changed to better fit our idea on how MPLS should be integrated into the network stack. Unlike Ayame it was our goal to reduce the impact of the rest of the network stack to a minimum.Especially no changes to the INET and INET6 should be needed.

netmpls

The MPLS stack itself is located in */sys/netmpls/*:

- *mpls.h*
  Header file defining MPLS structures and values. Most important the `sockaddr_mpls` and the SHIM header are defined here.
- *mpls_input.c*
  MPLS input processing, the netisr for MPLS is defined here. Additionally functions for TTL adjustment and error handling reside in this file.
- *mpls_output.c*
  MPLS output processing including forced packet checksumming and TTL inheritance from the IP header. Since no network driver supports delayed checksumming of MPLS packets it is necessary to update all checksums before sending out an MPLS packet.
- *mpls_proto.c*
  Definition of the various protocol structures to correctly hook MPLS into the network stack.
- *mpls_raw.c*
  sysctl and protosw implementation. It would be possible for userland to use a `AF_MPLS`, `SOCK_RAW` socket to send MPLS packets but this is currently untested and not used at all.
- *mpls_shim.c*
  Functions implementing the push, pop and swap operation needed by the input and output processing.

The code in *netmpls* is small, all in all 1200 lines of code including comments.

Routing Table

The MPLS stack itself can only be so small because it is able to use already available APIs to do a lot of work. For example it was important to use the already existing routing table and routing socket code for MPLS. While it is trivial to have an additional address family dependent routing table for MPLS it was more tricky to make the rtsock support MPLS. This is because not only the MPLS specific routing table needs to be updated with MPLS information but routing entries of other address families need MPLS information attached to them. It is possible to do this in two ways:

1. Do a second MPLS specific lookup on output processing to obtain the needed label information.
2. Attach the MPLS label information to the route entry in a similar way that the L2 MAC addresses are stored in the routing table.

We chose to use the available `rt_llinfo` entry in the routing table to store the MPLS label information on the routes and skipping the second lookup. The only problem is that now we need to suddenly add, remove and change MPLS information on already existing routes without causing havoc. For this a special `rtmsg` flag `RTF_MPLS` was added. Setting this flag on an `rtmsg` instructs the kernel to only update the MPLS information. While messages without this flag will clear the MPLS information if the nexthop of the route is changed. Getting these interactions on the routing socket correct was one of the biggest issues.

Data Link Layer

As mentioned MPLS is somewhere between OSI layer 3 and layer 2. In OpenBSD MPLS is currently supported on Ethernet, gre(4), gif(4), and lo(4) interfaces. In the case of Ethernet a special Ethernet type `0x8847` was defined for MPLS packets. MPLS input handling was added to `ether_input()`. In `ether_input()` and on all other input functions MPLS packets are pushed onto the MPLS netisr input queue and an MPLS netisr is scheduled. While the input processing is trivial output processing isn't. The MPLS output function needs to be called somewhere between Layer 3 and Layer 2. In Ayame the MPLS handling was done at the end of the Layer 3 in `ip_output()` more or less right before calling the interface output function. At the beginning we opted for doing it at Layer 2 since at least some changes were needed to support MPLS in the output functions whereas non were needed in `ip_output()` but this caused troubles because the output functions were suddenly called recursively. So realizing that neither Layer 3 nor Layer 2 are the right location we came up with a 3rd solution. `ifp->if_output()` is the link

layer specific output function. Now if this function pointer would be replaced with an MPLS specific output function then `mpls_output()` would be called right between Layer 3 and Layer 2 as it has to be. Another benefit from this is that MPLS can now be enabled on a per interface base. The output routine is swapped when the `IFXF_MPLS` flag is set on an interface.

mpe(4)

The mpe(4) interface is an MPLS Provider Edge pseudo-device used to connect a rdomain to an MPLS cloud. On most systems packets entering a VRF just appear which makes it fairly hard to filter or traffic shape such traffic. Because of this mpe(4) was developped. mpe(4) is a pseudo interface from where all traffic for a certain FEC will go over when entering or exiting the MPLS network. Since this is a real interface both pf(4) and altq(4) can operate on it. It is even possible to use bpf(4) on the mpe(4) interface to see the traffic in tcpdump. For BGP MPLS VPNs bgpd(8) will query the mpe(4) interface to get the label identifying this endpoint. Traffic routed to the mpe(4) interface will be added by bgpd with the necessary label information attached to the route. So that the generated outgoing packet is sent out with the correct MPLS label stack.

ldpd(8)

ldpd(8) is based on the same 3 process design as all other routing daemons in OpenBSD. It is actually based on ospfd(8). The privileged parent process is responsible to talk to the kernel and update the routing table. The two unprivileged children -- the ldpe and lde -- run chrooted. All processes use the IMSG framework to communicate with each other and the ldpe also has a UNIX domain socket as an interface for ldpctl(8). Some of the ospfd(8) inheritance is still obvious but LDP is very different from the OSPF protocol. LDP messages are mostly TLV encoded and there is no distributed DB to sync or DR and BDR to elect on a network. Instead LDP opens a session with each neighboring LDP router. In this regard LDP is a bit like BGP.

ldpe

The ldp engine (ldpe) is the process that talks to the network. Its purpose is to send out periodic hello packets and to establish sessions to other LDP neighbors. LDP is a a bit strange since it uses multicast hello packets like ospfd to find other LDP routers but it initiates TCP sessions like bgpd to each neighbor found. Once a session is opened a parameter exchange happens and if everything succeeds the session is established and it is now possible to distribute labels between the systems. The ldpe will parse the various TLV messages and send the extracted information to the lde. Like the bgpd(8) session engine it is the ldpe that sends out the periodic keep-alive messages.

lde

The label distribution engine (lde) is the process responsible to build the label information base. It gets various messages from the ldpe with the label mappings and address information of the neighbor systems. It also receives routing informations from the parent process and with these informations the label information base is constructed. Local incoming labels are assigned to FEC and the outgoing labels of the neighbors are attached to the FEC together with their nexthop information. All in all the protocol would not be very complex if the IETF did not add multiple buttons to the distribution process to make it more flexible and much more complex. The behaviour of the lde depends on three buttons. First of all the distribution behaviour. This can be either ordered or independent. When using ordered distribution of labels ldpd must wait until it has a valid outgoing mapping for the FEC before distributing the incoming label to other neighbors. The second button changes the way advertisements are made. In unsolicited mode labels are flooded to all peers but in ondemand mode every neighbor needs to request labels explicitly per FEC. Finally the retention of labels can be changed as well. In liberal retention all labels to all FEC are stored even if they are currently invalid or unused. In conservative retention mode all labels that are currently unused are freed and need to be requested when needed. In most cases liberal retention with independent distribution and unsolicited advertisements is used and this is the mainly tested mode of ldpd(8).

parent

As in ospfd and bgpd the parent process main work is to keep the kernel routing table in sync with the label information base of the lde. It also tracks the link states and interface address changes that are used by the ldpe. Unlike routing protocols LDP does not track the reachability of routes. It just assigns labels to each route or FEC and tries to build LSPs to these destinations.

bgpd(8)

Configuring MPLS VPNs requires multiple steps. First of all a routing domain for a customer needs to be setup. Then a mpe(4) interface needs to be created in this rdomain and configured. Everything else can be configured or figured out by bgpd(8). To make

bgpd(8) support BGP MPLS VPNs a few things had to be implemented first. First of all extended community attributes had to be supported. The extended community attributes are used to tag and identify routes that need to be imported into a particular rdomain. These are the so called import and export targets. Additionally a major redesign of the kroute code was needed so that it is possible to handle multiple routing tables at the same time in bgpd. Finally supporting the new SAFI (subsequent address family) was fairly simple since bgpd was already mostly address independent.

Example

The following example shows a simple MPLS enabled network with two PE routers and four P routers.

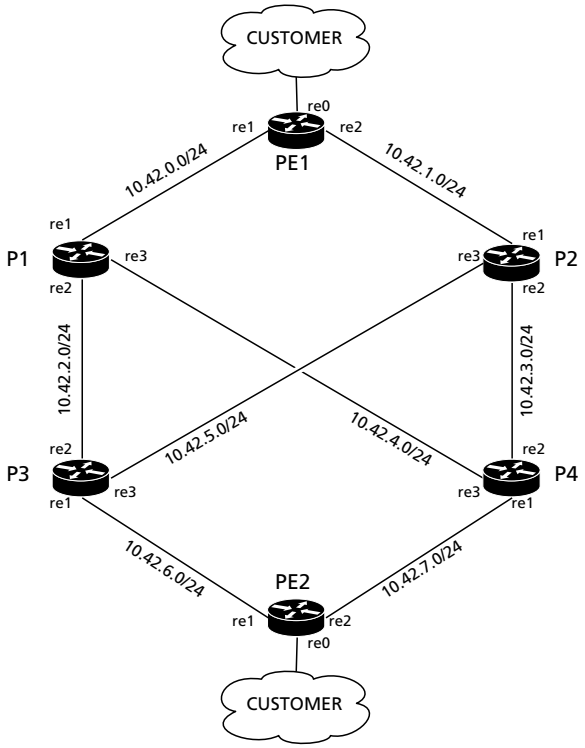

Figure 2: Example MPLS network

Here the interface configuration of the P routers.

Interface config P1

```
hostname P1
ifconfig lo1 10.42.21.1/32
ifconfig re1 10.42.0.2/24 mpls
ifconfig re2 10.42.2.1/24 mpls
ifconfig re3 10.42.4.1/24 mpls
```

Interface config P2

```
hostname P2
ifconfig lo1 10.42.21.2/32
ifconfig re1 10.42.1.2/24 mpls
ifconfig re2 10.42.3.1/24 mpls
ifconfig re3 10.42.5.1/24 mpls
```

Interface config P3

```
hostname P3
ifconfig lo1 10.42.21.3/32
ifconfig re1 10.42.6.1/24 mpls
ifconfig re2 10.42.2.2/24 mpls
ifconfig re3 10.42.5.2/24 mpls
```

Interface config P4

```
hostname P4
ifconfig lo1 10.42.21.4/32
ifconfig re1 10.42.7.1/24 mpls
ifconfig re2 10.42.3.2/24 mpls
ifconfig re3 10.42.4.2/24 mpls
```

All Ethernet interfaces are MPLS enabled and each P router has a loopback address assigned which will be used in the ldpd and ospfd config. Both the *ldpd.conf* and *ospfd.conf* files are the same on all P routers with the exception of the *router-id* which is set to the loopback interface address of *lo1*.

ospfd.conf of router P1

```
router-id 10.42.21.1

area 0.0.0.0 {
        interface re1
        interface re2
        interface re3
        interface lo1
}
```

The *ospfd.conf* just enables OSPF on all interfaces, nothing more is needed for this basic setup.

ldpd.conf of router P1

```
router-id 10.42.21.1

interface re1
interface re2
interface re3
```

The *ldpd.conf* enables LDP on all the MPLS enabled interfaces. Apart from enabling IP forwarding and starting ospfd and ldpd nothing more needs to be done on the P routers.

The configuration on the PE routers is a bit more complex.

Interface config PE1

```
hostname PE1
ifconfig re0 rdomain 1
ifconfig re0 192.168.237.2/28
route -T1 add default 192.168.237.1
ifconfig lo1 10.42.42.1/32
ifconfig re1 10.42.0.1/24 mpls
ifconfig re2 10.42.1.1/24 mpls
ifconfig mpe0 rdomain 1
ifconfig mpe0 mplslabel 666
ifconfig mpe0 192.168.237.2/32
```

re1 and *re2* are MPLS enabled and *re0* is moved into *rdomain* 1 to connect to the customer network. Additionally there is a *mpe0* in the same *rdomain* 1 and will act as the edge device for this customer. The config on PE2 is more or less equal.

Interface config PE2

```
hostname PE2
ifconfig re0 rdomain 1
ifconfig re0 192.168.237.242/28
ifconfig lo1 10.42.42.2/32
ifconfig re1 10.42.6.2/24 mpls
ifconfig re2 10.42.7.2/24 mpls
ifconfig mpe0 rdomain 1
ifconfig mpe0 mplslabel 666
ifconfig mpe0 192.168.237.242/32
```

The ospfd and ldpd configuration on the PE routers are very similar to the ones on the P routers. Only the list of interfaces is a bit different.

ospfd.conf of router PE1

```
router-id 10.42.42.1

area 0.0.0.0 {
        interface re1
        interface re2
        interface lo1
}
```

ldpd.conf of router PE1

```
router-id 10.42.42.1

interface re1
interface re2
```

The last bit missing is the configuration of bgpd. First the settings for *rdomain* 1 are defined. The values of the *rd*, *import-target* and *export-target* are the same in this example but they don't need to be the same. The *rd* needs to be a unique value accross all configured VPNs in case two customers have the same addresses configured. The *import-target* communities need to match the *export-target* communities on the other PE routers. It is possible to set multiple *import-targets* and *export-targets* per *rdomain*.

bgpd.conf of router PE1

```
router-id 10.42.42.1
AS 3.10

rdomain 1 {
        descr "CUSTOMER1"
        rd 3.10:1
        import-target rt 3.10:1
        export-target rt 3.10:1
        depend on mpe0
        network inet connected
        network 0.0.0.0/0
}

group ibgp {
        announce IPv4 unicast
        announce IPv4 vpn
        remote-as 3.10
        local-address 10.42.42.1
        neighbor 10.42.42.2 {
                descr PE2
        }
}
```

The second part of the config creates the session between PE1 and PE2. Both IPv4 unicast - the default table - and the IPv4 VPN subsequent address family are enabled on the session to PE2. The session are setup on the loopback IPs of both PE routers. Here the PE2 *bgpd.conf* to complete the configuration.

bgpd.conf of router PE2

```
router-id 10.42.42.2
AS 3.10

rdomain 1 {
        descr "CUSTOMER1"
        rd 3.10:1
        import-target rt 3.10:1
        export-target rt 3.10:1
        depend on mpe0
        network inet connected
}

group ibgp {
        announce IPv4 unicast
        announce IPv4 vpn
        remote-as 3.10
        route-reflector
        local-address 10.42.42.2
        neighbor 10.42.42.1 {
                descr PE1
        }
}
```

Future work

The OpenBSD MPLS support seems to be enough stable and compliant to setup BGP MPLS VPN tunnels with other systems. Quite a bit of testing was done against systems from Cisco and other OpenBSD users were able to run LDP sessions to Juniper boxes. The kernel infrastructure seems to have stabilized and so the current focus on the development are missing features - for example PWE3 and VPLS support. Additionally a lot of work is still needed on ldpd(8) certain features and operation modes are not fully implemented yet. The third big area that needs some work is the PE - CE interaction. Making it possible to import and export OSPF LSDB entries through BGP is probably the biggest challenge in this area.

References

[1] Multiprotocol Label Switching Architecture, RFC 3031, January 2001.

[2] MPLS Label Stack Encoding, RFC 3032, January 2001.

[3] LDP Specification, RFC 3036, January 2001.

[4] BGP Extended Communities Attribute, RFC 4360, February 2006.

[5] BGP/MPLS IP Virtual Private Networks (VPNs), RFC 4364, February 2006.

[6] Avoiding Equal Cost Multipath Treatment in MPLS Networks, RFC 4928, June 2007.

[7] AYAME project, http://www.ayame.org/