A networking powerhouse

QEMU, VDE and Dnsmasq

In another Wiki article I talk about using the virtualization software called QEMU as an Open Source alternative to the commercial / closed source VMWare. QEMU needs a little help and tweaks of your computer environment to give it some of the capabilities VMWare has. In this Wiki page I will show you how to use the power of Linux software collaboration and setup a network environment where you can use your Virtual Machine inside QEMU as if it were physically connected to your network. Running a guest Operating System like MS Windows in the QEMU machine virtualizer on your Slackware host is really cool, and for some people this is an opportunity to get away from the requirement to have a Windows desktop running all the time. With QEMU's “user-mode” network stack, the Guest OS has network access to the host computer and to the network beyond. When user-mode network is activated, QEMU runs it's own internal DHCP server which provides a guest OS with a single IP address if the Guest is configured to use DHCP.

However, when your networking demands are higher, user-mode networking just is not enough.
The drawback of user-mode networking is that is is based on the SLIRP protocol which does not allow for UDP connections, plus it's outbound only, meaning that no one on your network, not even the Host Operating System, will be able to connect to the Guest OS. This isolates the Guest and this might not be desirable if you want to provide functionality on your network that is supposed to run inside QEMU.

Another example: you are running several QEMU virtual machines in parallel, and the Guest OS-es need to communicate with each other - this is impossible with user-mode networking. To be truthful, this was impossible until version 0.8.0 of QEMU. The 0.8.0 release was the first to support *VLAN*, an elegant but limited way to connect multiple Guest OS-es to each other. But even with the *vlan* option added, the Host OS will still not be able to connect to any of the Guests.
QEMU offers a slightly more advanced way of bridged networking using a *tap* device, so that two-way communication between Host and Guests becomes possible, but the disadvantages of using a tap device with QEMU are, that you have to setup a separate tap device for each Virtual Machine, and root access (or `sudo`) is required for connecting the VM to the network device.

VDE

The solution is a small but powerful piece of software called VDE - which stands for *Virtual Distributed Ethernet*. You can see VDE as the software incarnation of a hardware network switch plus attached cables. Using the *vde_switch* and *vde_plug* programs you are able to create quite complex virtual analogies of a network that can span several hosts, even across the Internet. Read the documentation if you want to know more about this exciting product. For our limited purpose of enhancing the network capabilities of the QEMU Virtual Machines, I will just point out that VDE configures a *virtual* network for your QEMU Guests, and uses a single tap device to set it all up. You configure the tap device as root (typically during the boot stage of your computer) after which no further root access is required for the Virtual Machines to connect to the VDE “switch”.

Getting a VDE package

I have created a Slackware package for VDE that you'll find in my SlackBuilds repository. The package

by the way installs a version of the example rc.vdenetwork script in it's *doc* directory.

DNSMasq

From the Dnsmasq homepage:
"Dnsmasq is a lightweight, easy to configure DNS forwarder and DHCP server. It is designed to provide DNS and, optionally, DHCP, to a small network. It can serve the names of local machines which are not in the global DNS. The DHCP server integrates with the DNS server and allows machines with DHCP-allocated addresses to appear in the DNS with names configured either in each host or in a central configuration file. Dnsmasq supports static and dynamic DHCP leases and BOOTP for network booting of diskless machines.
Dnsmasq is targeted at home networks using NAT and connected to the internet via a modem, cable-modem or ADSL connection but would be a good choice for any small network where low resource use and ease of configuration are important."

I could not have said it better myself. We will use dnsmasq to compensate for the loss of the QEMU-internal DHCP Server (because we abandon user-mode networking). Note that a Slackware full-install contains a dnsmasq package already, and a `/etc/rc.d/rc.dnsmasq` init script is installed automatically, although it is not executable by default. If your machine already runs dnsmasq, you will have to look at the `rc.vdenetwork` script in the last section and integrate the dnsmasq related configuration into your existing setup. If you're *not* already running dnsmasq, you don't have to worry about this `rc.dnsmasq` file that Slackware installed - we won't be using it.

Dnsmasq will pick up any existing network configuration information of your computer, by reading `/etc/hosts` and `/etc/resolv.conf` and will use that information when it sends DHCP configuration to the QEMU Guest. If your computer has a functioning network connection to the LAN and/or the Internet, your QEMU Guest(s) will enjoy the same functionality. Furthermore, by examining your computer's message log `/var/log/messages` you will be able to determine what IP address the QEMU Guest picks up, and you can use that knowledge to `ssh` into the Virtual Machine or point a web browser to a running Apache server inside the VM - or connect to whatever other network service that you enabled on your VM.

Tying it all together

This section documents how to combine the functionality of vde, dnsmasq and iptables and achieve what we set out to do: have better, no-fuss networking functionality for QEMU.

How it works

- First of all, we load the "tun" driver which is a kernel module that you should have enabled for your kernel. Slackware kernels have this tun module, if you compile your own kernels, make sure you configure

```
CONFIG_TUN=m (Device Drivers - Network device support -> Universal
TUN/TAP device driver support)
CONFIG_IP_NF_CONNTRACK=m (Connection tracking)
CONFIG_IP_NF_IPTABLES=m (IP tables support)
```

```
CONFIG_IP_NF_NAT=m (Full NAT)
```

Instead of building these as modules, you can of course compile them into the kernel by changing the 'm' to 'y'. Also, the device `/dev/net/tun` needs to exist. On my Slackware system it is created automatically when I run

```
modprobe tun
```

(or let the `rc.vdenetwork` init script below load the module). If your system does not automatically create that device file, you should create it yourself:

```
mkdir /dev/net
mknod /dev/net/tun c 10 200
chmod 660 /dev/net/tun
```

- The tap0 interface is used by the program *vde_switch* which will do all the hard work of managing the virtual network connections of the VM's that connect to it through the TAP device. So we start vde_switch:

```
vde_switch -tap tap0 -daemon
```

and we will allow any user program to connect to and use the control socket:

```
chmod -R a+rwx /var/run/vde.ctl
```

- Then we configure the resulting *tap0* network interface with an IP address. The QEMU Virtual Machines that you'll be running will all be available on a subnetwork that is “behind” this *tap0* device.

```
ifconfig tap0 10.111.111.254 broadcast 10.111.111.255 netmask
255.255.255.0
```

The TCPIP configuration values I used in this example are arbitrary - you may choose whatever you like, as long as you pick an address within a subnet range that is not used on your local network. The Virtual Machines you are going to be running will all be in the IP subnet defined by the *tap0*'s IP address and netmask. The tap0 will act as the default gateway for that subnet.

- Traffic to and from the subnet behind the tap0 interface must be forwarded

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

and since we are most probably setting up all of this on an ordinary workstation, we don't want to cause network disruption by suddenly announcing the birth of a new network segment. We will “hide” the tap0 interface and our QEMU/VDE subnet behind a firewall, by applying couple of NAT *iptables* rules. Tus, we designate the eth0 interface as the external firewall interface (in the example I use *eth0* - your external interface might be called different).

```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

- We've setup the foundation now, and need to finish by starting *dnsmasq* to have a DHCP/DNS server in case the QEMU Guest OS needs these:

```
/usr/sbin/dnsmasq \
          --log-queries \
          --user=named \
          --dhcp-leasefile=/var/state/dhcp/qemu-dhcpd.leases \
          --dhcp-
range=10.111.111.129,10.111.111.199,255.255.255.0,10.111.111.255,8h \
          --interface=tap0 \
          --domain=qemu.lan
```

  This is quite a long command line, but don't worry - if you're going to use that example `rc.vdenetwork` script below, you don't have to type it all the time  What this does, is setup a range of IP addresses to be used for DHCP. These addresses are all in the subnetwork range defined by the IP address of the *tap0* interface. The command will enable the DHCP server only on the internal *tap0* interface so that the external eth0 (or whatever) interface is not affected. You could seriously mess up your network if suddenly a second DHCP server started feeding weird IP addresses to the computers in your LAN (assuming you do have a DHCP server in the LAN of course - most households with Cable/DSL internet run a DHCP server on their Internet router).

- We're done! If you're running a 2.4 kernel and get the following warning message when starting QEMU:

```
Could not configure '/dev/rtc' to have a 1024 Hz timer. This is not a
fatal
error, but for better emulation accuracy either use a 2.6 host Linux
kernel
or type 'echo 1024 > /proc/sys/dev/rtc/max-user-freq' as root.
```

  you'll have to do just that: run

```
echo 1024 > /proc/sys/dev/rtc/max-user-freq'
```

  as root.

Implementation

To activate your VDE powered virtual ethernet expansion every time the computer boots, you add an *init* script to Slackware's `/etc/rc.d` directory and make sure the script gets called when Slackware starts. The last section contains a script that you can save as `/etc/rc.d/rc.vdenetwork`. This script will perform all the stuff I talked about just now, setting up everything you need to combine the functionality of QEMU, VDE and dnsmasq.

- Use the example in the last section to create a file that you call `/etc/rc.d/rc.vdenetwork`. Edit the file to make sure that it meets your requirements (see a few lines below for an explanation of the variables and values used):

```
vi /etc/rc.d/rc.vdenetwork
chmod +x /etc/rc.d/rc.vdenetwork
```

The second command will make the script executable, so that we can call it from `rc.local`.

- Add the following lines to the file `/etc/rc.d/rc.local`:

```
if [ -x /etc/rc.d/rc.vdenetwork ]; then
  # echo "VDE network:    /etc/rc.d/rc.vdenetwork start"
  /etc/rc.d/rc.vdenetwork start
fi
```

and Slackware will run the `rc.vdenetwork` script for you next time you boot the machine.

- Important variables in the `/etc/rc.d/rc.vdenetwork` script that you might want to examine and modify before running it for the first time (extracted from the script below):

```
TAP_DEV=tap0                    # The name of the TAP interface you'll use
TAP_IP=10.111.111.254           # The IP address you want to give to the
TAP device
TAP_MASK=255.255.255.0          # The TAP device's netmask
VM_IPLOW=10.111.111.128         # The start IP address of DNSmasq's DHCP
range
VM_IPHIGH=10.111.111.199        # The end IP address of DNSmasq's DHCP
range
                                # The VM_IPLOW an VM_IPHIGH addresses must
all lie within
                                # the same IP subnetwork, defined by
TAP_IP and TAP_MASK
VM_DOMAIN=qemu.lan              # DNSmasq will create a DNS domain for
your Virtual Machines.
NAT_IFS="eth+"                  # Any other interface(s) present in the
host computer
                                # that should be configured as external
NAT interfaces.
                                # If you have a eth0 and wlan0 device, you
could use
                                # NAT_IFS="eth0 wlan0". The plus sign in
"eth+" means
                                # "all network interfaces starting with
'eth'"
DNSMASQ_OPTIONS=""              # Any other options you want to pass to
dnsmasq.
```

- You can now reboot, or if you don't want to reboot, just run

```
/etc/rc.d/rc.vdenetwork start
```

We're all set! It is time to start your QEMU session and see if it all works as expected! I have provided another script in the last section of this article as an example of how to run QEMU and

have it using VDE for the networking.

Example scripts

Save this code as `/etc/rc.d/rc.vdenetwork`

```
#!/bin/sh
# QEMU/VDE/DnsMasq environment preparation script
# ------------------------------------------------------------------------
-
#
# After running this startup script, run a QEMU virtual machine in this way:
#
#     vdeqemu [qemu_option [qemu_option], ...]
#
# The vdeqemu program will automatically connect
# the QEMU virtual machine to the VDE switch.
#
# ------------------------------------------------------------------------
-

# The IP configuration for the tap device that will be used for
# the virtual machine network:

TAP_DEV=tap0
TAP_IP=10.111.111.254
TAP_MASK=255.255.255.0

TAP_BCAST=`/bin/ipmask ${TAP_MASK} ${TAP_IP} | cut -f 1 -d ' '`

# Host interfaces that need to be NAT-ed (in case we're not bridging):
NAT_IFS="eth+"

# Definitions for the LAN segment the Qemu virtual machines will be in.
# These definitions will be fed to dnsmasq - this program will provide DNS
# and DHCP to the Qemu LAN.

# The VM_IPLOW and VM_IPHIGH addresses must agree with the definitions for
# the tap0 device above. These 'low' and 'high' values are the IP address
# range for the DHCP server to use.

VM_DOMAIN=qemu.lan
VM_IPLOW=10.111.111.128
VM_IPHIGH=10.111.111.199
VM_BCAST=${TAP_BCAST}
VM_MASK=${TAP_MASK}

# For additional options to dnsmasq:
#DNSMASQ_OPTIONS="--server /my.net/192.168.1.1"
```

```
DNSMASQ_OPTIONS=""

# See how we were called.

case "$1" in
  start)
        echo -n "Starting VDE network for QEMU: "

        # Load tun module
        /sbin/modprobe tun 2>/dev/null
        # Wait for the module to be loaded
        while ! /bin/lsmod |grep -q "^tun"; do echo Waiting for tun
device;sleep 1; done

        # Start tap switch
        vde_switch -tap ${TAP_DEV} -daemon

        # Bring tap interface up
        ifconfig ${TAP_DEV} ${TAP_IP} broadcast ${TAP_BCAST} netmask
${TAP_MASK}

        # Start IP Forwarding
        echo "1" > /proc/sys/net/ipv4/ip_forward
        for NIC in ${NAT_IFS}; do
          iptables -t nat -A POSTROUTING -o ${NIC} -j MASQUERADE
        done

        # Change pipe permission (vde2 uses a different pipe directory)
        if vde_switch -v | grep -q "^VDE 1" ; then
          chmod 666 /tmp/vde.ctl
        else
          chmod -R a+rwx /var/run/vde.ctl
        fi

        # If we are not running 2.6, apply workaround
        if uname -r | grep '^2.4'; then
          echo 1024 > /proc/sys/dev/rtc/max-user-freq
        fi

        # Start dnsmasq, the DNS/DHCP server
        # for our Virtual Machines behind the tap0 interface
        /usr/sbin/dnsmasq \
          --log-queries \
          --user=named \
          --dhcp-leasefile=/var/state/dhcp/qemu-dhcpd.leases \
          --dhcp-range=${VM_IPLOW},${VM_IPHIGH},${VM_MASK},${VM_BCAST},8h \
          --interface=${TAP_DEV} \
          --domain=${VM_DOMAIN} \
          $DNSMASQ_OPTIONS
        echo
        ;;
```

```
  stop)
        echo -n "Stopping VDE network for QEMU: "
        {
        # Delete the NAT rules
        for NIC in ${NAT_IFS}; do
          iptables -t nat -D POSTROUTING -o ${NIC} -j MASQUERADE
        done
        # Bring tap interface down
        ifconfig ${TAP_DEV} down
        # Kill VDE switch
        pgrep -f vde_switch | xargs kill -TERM
        # Remove the control socket
        rm -f /tmp/vde.*
        rmdir /var/run/vde.ctl
        # Stop dnsmasq
        pgrep -f dnsmasq | xargs kill -TERM
        } >/dev/null 2>&1
        echo
        ;;
  restart|reload)
        $0 stop
        sleep 1
        $0 start
        ;;
  *)
        echo "Usage: $0 {start|stop|restart|reload}"
        exit 1
esac
```

A script that you can use to start QEMU, connect it to the vde_switch, and have sound in the VM is presented here. Note that if you run more than one QEMU session, the Virtual Machines will see each other on the network provided by the VDE switch. This means that all of them must have unique MAC addresses. Since QEMU will assign the *same* MAC address to each VM by default, we will have to pass each QEMU instance it's own MAC Address. So, for running multiple QEMU powered VM's, you'll have to create multiple copies of the following script (or think up some magic to generate unique MAC addresses). Actually, I also provide this same example script on the QEMU Wiki page. The example assumes you want to run Windows XP, so that explains the comments and the naming of the various files used. You can run anything you want inside the QEMU VM or course, QEMU won't care.

```
#!/bin/sh
#
# Start Windows XP Pro in QEMU using VDE for better network support

PARAMS=$*

# Qemu can use SDL sound instead of the default OSS
export QEMU_AUDIO_DRV=sdl

# Whereas SDL can play through alsa:
export SDL_AUDIODRIVER=alsa
```

```
# Change this to the directory where _you_ keep your QEMU images:
IMAGEDIR=/home/alien/QEMU

# Change this to the directory where _you_ keep your installation CDROM's
ISO images:
ISODIR=/home/alien/ISOS

# Now, change directory to your image directory
cd $IMAGEDIR

# If you want to boot from the WinXP CD add a '-boot d' parameter to the
commandline;
#   if you don't need the CDROM present in the VM, leave '-cdrom
${ISODIR}/winxp_pro_us.iso' out:
# I made the MAC address up - make sure it is unique on your (virtual)
network.

# This command returns to the command prompt immediately,
#   and QEMU's (error) output is redirected to logfiles.
vdeqemu -net vde,vlan=0 -net nic,vlan=0,macaddr=52:54:00:00:EE:02 -m 256 -
localtime -soundhw all -hda winxp.img -cdrom ${ISODIR}/winxp_pro_us.iso
1>winxp.log 2>winxp.err ${PARAMS} &
```