LXC Containers

Dr. **Jose L. Muñoz Tapia**
Information Security Group (ISG)
Universitat Politècnica de Catalunya (UPC)

LXC

Outline

Introduction

- The standard tools for complete Linux containers is LXC.
- Complete containers are like virtual machines with all its relevant environment isolated and being able to execute multiple processes inside.
- LXC can be installed as (ubuntu 16.04):

```
hypervisor# apt install lxc1
```

- In ubuntu 18.04 install also the templates for vritual machines:

```
hypervisor# apt install lxc-templates
```

Create i

- To create a container with LXC:

```
hypervisor# lxc-create -n mycontainer -t ubuntu
```

- The `-n` or `--name` option specifies the name of the container.
- The `-t` option specifies the template to create the filesystem of the container:
  - An LXC template is nothing more than a script which builds a container for a particular Linux environment.
  - By default, templates reside in /usr/lib/lxc/templates on Ubuntu.
  - The templates download the data to create a system.

Create ii

- Note. For Debian-based systems the template uses a utility called `debootstrap`.
- If you get an error about `lxc-create` or `debootstrap` make sure you installed the lxc package.

- By default, it will create a minimal Ubuntu install of the same release version and architecture as the local host:
  - If you want, you can create Ubuntu containers of any arbitrary version by passing the release parameter.
  - For example, to create a Ubuntu 14.10 container:

```
hyperv# lxc-create -n <cont-name> -t ubuntu -- --release utopic
```

Create iii

- After a series of package downloads and validation, an LXC container image are finally created:
  - You will see a default login credential to use.
  - The container is stored in /var/lib/lxc/<container-name>.
  - Its root filesystem is found in /var/lib/lxc/<container-name>/rootfs.
  - All the packages downloaded during LXC creation get cached in /var/cache/lxc, so that creating additional containers with the same LXC template will take no time.

SELinux

- New linux distros apply some security mechanisms that support control security policies for applications.
- These are called Security-Enhanced Linux (SELinux).
- The solution to SELinux in Ubuntu is called App Armor.
- App Armor can cause problems to LXC containers.
- To overcome this issue you have to add the following line to the container's config file (/var/lib/lxc/mycontainer/config):

```
lxc.aa_profile = unconfined
```

Start/Stop i

- To start the container type the following command:

```
hypervisor# lxc-start -n mycontainer
```

- Once you do this you will be prompted with a login and asking for your username and password.
- This, by default, is ubuntu for both.
- In the console of the container, you can shut down it as any other Linux box:

```
mycontainer# halt
```

- You can also shut down the container from the **hypervisor**:

```
hypervisor# lxc-stop -n mycontainer
```

Start/Stop ii

- Use `-d` or `--daemon` to ”daemonize“ the container:

```
hypervisor# lxc-start -n mycontainer -d
```

- To connect to a container that has been started in the background, you can use the following command:

```
hypervisor# lxc-console -n mycontainer
```

- You can exit the console of a container without shutting down it by typing `ctrl+a` then `q`.

Start/Stop iii

- The command `lxc-wait` waits for a specific container state before exiting:
  - This command is very useful in scripting.
  - In this example the command will terminate when the specified container reaches the state RUNNING:

```
hypervisor# lxc-wait -n mycontainer -s 'RUNNING'
```

Destroy

- The command `lxc-destroy` completely destroys the container and removes it from your system.
- The required argument is the container name but you can also pass along -f to tell LXC to force a deletion if the container is currently running (default is to abort/error out).
- Example:

```
hypervisor# lxc-destroy -n mycontainer
```

Freeze

- We can also freeze or pause a container.
- The command `lxc-freeze` and `lxc-unfreeze` lets you pause the container (stop all processes from running).
- lxc-unfreeze will thaw all the processes previously frozen.

Get information i

- There are several tools that come with the LxC package to get information about containers.
- For example, with `lxc-ls` we can see the containers that are running, stopped, frozen or active.
- With the option `--fancy` basic information is displayed:

```
hypervisor# lxc-ls --fancy
hypervisor# lxc-info -n mycontainer
```

- `lxc-checkconfig` checks the current kernel for LxC support.

Get information ii

- The command `lxc-info` is similar to `lxc-ls` but requires the -n <container name> argument:
  - It tells you the state of the container as well as its process ID on the `hypervisor` and information about resources used.
  - This command will also tell you if the container is frozen.
- The command `lxc-monitor` monitors the state of containers.
- By default the output is not saved in a log, but with `--logfile=FILE` parameter we can choose where to save the log.

Attach: Run a Command

- The command `lxc-attach` let us run a command in a running container.
- This command is mainly used when you want to quickly launch an application in an isolated environment or create some scripts.
- Example:

```
hypervisor# lxc-attach -n webserver -- ifconfig eth1 192.168.1.2/24
```

- Using the option `-s namespaces` allows you to specify the namespaces to attach to, as a pipe-separated list:
  - Example: NETWORK|IPC.
  - Allowed values are MOUNT,PID,UTSNAME,IPC,USER and NETWORK.
  - This allows one to change the context of the process to e.g. the network namespace of the container while retaining the other namespaces as those of the host.
  - Important: this option needs to use also the `-e` option (elevated privileges).

Limiting Resources

- The command `lxc-cgroup` allows you to set inner workings of a specific container (these can also be set in the container's "config" file as well).
- The one that most people will probably find helpful is setting memory limits.
- This can be done like this (300M RAM limit):

```
hypervisor# lxc-cgroup -n mycontainer memory.limit_in_bytes 300000000
```

Devices

- The command `lxc-device` let us manage devices in running containers.
- At this point, only the `add` device action is supported.
- For example, we can create a device into a container based on the maching device on the **hypervisor**:

```
hypervisor# lxc-device -n mycontainer add /dev/sdb1
```

- The device (e.g. USB) will be available on the container.
- We can move the interface `eth0` from the **hypervisor** to a container as `eth3`:

```
hypervisor# lxc-device -n mycontainer add eth0 eth3
```

- To recover the interface back to the hypervisor:

```
hypervisor# lxc-attach -n mycontainer -e -s NETWORK -- ip link set eth3 1
```

Auto-start

- By default a new container will not start if the host system is rebooted.
- You can configure this in each container's configuration file:

```
lxc.start.auto = 1
lxc.start.delay = 5
```

- With these parameters:
  - The container will start when the host server boots.
  - Then the host system will wait 5 seconds before starting any other containers.
- We can also set a value for **`lxc.group`** parameter to group containers:
  - A container can belong to several groups by using this syntax:

```
lxc.group = group1,group2,group3
```

  - Then, with groups, we can easily shutdown, reboot, kill or list all the containers that belong to a specific group:

```
hypervisor# lxc-autostart -s -g <group>
```

Outline

Introduction

- By default, LXC simply stores the rootfs of containers under the directory /var/lib/lxc/<container>/rootfs.
- the `--dir` option can be used to override the path.
- However, apart from a simple directory with a traditional filesystem, LXC supports additional storage backends (also referred to as backingstores).
- These are overlayfs, btrfs, zfs and lvm.
- You can specify them in the `lxc-create` command with the `-B` option.
- For the btrfs and zfs, LXC can autodetect the backingstore and automatically use the underlying features of these filesystems.

Overlayfs i

- The overlayfs is used by default when cloning containers with the snapshot option:
  - It allows creating a container based on another one and storing only changes.
  - The Overlay-filesystem (or Unionfs) is a filesystem service that uses union mount.
  - Allows different filesystems hierarchies to appear as one unified filesystem.
  - The overlay-filesystem overlays one filesystem above the other into a layered representation.
  - When a directory appears in both layers, overlayfs forms a merged directory for both of them.

Overlayfs ii

- In case of two files have the same name in both layers:
  - Only one is served from the upper or the lower layer.
  - But if a file only exists in the lower layer and an edit needs to be done on that file, a copy of this file is created on the upper layer to be edited.
  - In most cases the lower layer is normally a read-only filesystem.
  - While the upper layer is read-write one.
  - This allows what is called **copy on write**.
- When used with lxc-create:
  - The overlayfs will create a container where any change done after its initial creation.
  - This will be stored in a “delta0” directory next to the container’s rootfs.

Btrfs, Zfs and LVM

- With btrfs, LXC will setup a new subvolume for the container.
- This makes snapshotting very easy.
- The zfs filesystem works in a similar way as btrfs does.
- With lvm, LXC will use a new logical volume for the container:
  - The LV can be set with `--lvname` (the default is the container name).
  - The VG can be set with `--vgname` (the default is ”lxc”).
  - The filesystem can be set with `--fstype` (the default is “ext4”).
  - The size can be set with `--fssize` (the default is “1G”).
  - You can also use LVM thinpools with `--thinpool`.

Copy clones

- A **copy** clone is a new container copied from the original.
- It takes as much space on the host as the original.
- To create copies, you can use the command `lxc-copy`.
- For example, to create a copy:

```
hypervisor# lxc-copy -n <original container name> -N <new container name>
```

Snapshot Clones i

- A **snapshot** is a read-only copy of a filesystem.
- Over a snapshot we can create a “snapshot copy container” which is a copy-on-write copy (which is writable).
- If you want to create a snapshot copy instead of a complete copy of a container you have to include the `-s` option in the command:

```
hypervisor# lxc-copy -s -n <original cont name> -N <new cont name>
```

- Remark that to create snapshots in LXC:
  - we cannot use a COW sparse file because the filesystem of containers is in a directory not in a file (like in User Mode Linux).
  - What we need are storage filesystems that allow the creation of a COW mount point so that changes to the container’s rootfs are properly stored.
  - The overlayfs, aufs, btrfs, zfs and LVM allow this functionality.
- In the particular case of snapshot copies of directory-packed containers, LXC creates them using the overlay filesystem:

Snapshot Clones ii

- For instance, a privileged directory-backed container C1 will have its root filesystem under /var/lib/lxc/C1/rootfs.
- A snapshot clone of C1 called C2 will be started with C1's rootfs mounted read-only under /var/lib/lxc/C2/delta0.
- Importantly, in this case C1 should not be allowed to run or be removed while C2 is running.
- It is advised instead to consider C1 a canonical base container, and to only use its snapshots.

Snapshots i

- A snapshot is a read-only copy of a container:
  - We can create snapshots for containers built with btrfs, lvm, zfs, and overlayfs.
  - To create snapshots, you use the command `lxc-snapshot`:

```
hypervisor# lxc-snapshot -n C1
```

  - The previous command creates a snapshot called 'snap0'.
  - The snapshot is stored under /var/lib/lxcsnaps.
  - The next snapshot will be called 'snap1' and so on.
  - Existing snapshots can be listed:

```
hypervisor# lxc-snapshot -L -n C1
```

- The container C1 can be modified and the snapshot is still consistent.
- A snapshot can be restored (erasing the current C1 container) using:

```
hypervisor# lxc-snapshot -r snap1 -n C1
```

Snapshots ii

- After the restore command:
  - The snap1 snapshot continues to exist.
  - And the previous C1 is erased and replaced with the snap1 snapshot.
- If lxc-snapshot is called on a directory-backed container:
  - An error will be logged and the snapshot will be created as a copy-clone (inconsistent if doing changes in the directory).
  - If snapshots of a directory backed container C1 are desired, then an overlayfs clone of C1 should be created.
  - C1 should not be touched again.
  - The overlayfs clone can be edited and snapshotted:

```
hypervisor# lxc-clone -s -o C1 -n C2
hypervisor# lxc-start -n C2 -d # make some changes
hypervisor# lxc-stop -n C2
hypervisor# lxc-snapshot -n C2
hypervisor# lxc-start -n C2
```

Ephemeral Snapshots

- While snapshots are useful for longer-term incremental development of images, **ephemeral containers** utilize snapshots for quick, single-use throwaway containers.
- Given a base container C1, you can start an ephemeral container using:

```
hypervisor# lxc-start-ephemeral -o C1
```

- The container begins as a snapshot of C1.
- Instructions for logging into the container will be printed to the console.
- After shutdown, the ephemeral container will be destroyed.

Outline

LXC Bridge Configuration

- The default systemd configuration in Ubuntu (16.04) is in the file: **/etc/default/lxc-net**.
- By default:
  - LxC creates a private network namespace for each container, which includes a layer 2 networking stack.
  - LxC also automatically creates a network bridge called `lxcbr0` that is started when the host is started.
  - The bridge interface is NATed to enable the containers to access the outside world with a veth endpoint passed into the container.
  - The NAT is created with `iptables` with a rule to basically masquerade all the traffic leaving the containers which are bridged with `lxcbr0`.
  - By default, the LXC installation also launches a `dnsmasq` process that acts as DHCP server for `lxcbr0`.
  - This DHCP server uses IPv4 addresses from 10.0.3.2 to 10.0.3.254.

DNS Resolution i

- We can enable DNS resolution between containers and from the hypervisor (host, which is a Ubuntu 16.04).
- To do so, we install a system-wide dnsmasq in the hypervisor:

```
hyp# apt install dnsmasq
```

- We will have two dnsmasq:
  1. One is the “lxc dnsmasq”, which is started by the service lxc-net.
  2. The other is a “system-wide lxc”, which is started by the package dnsmasq.

DNS Resolution ii

- To configure the system-wide dnsmasq edit /etc/dnsmasq.d/lxc as follows:

```
# Tell any system-wide dnsmasq instance to make sure to bind to interfaces
# instead of listening on 0.0.0.0
# WARNING: changes to this file will get lost if lxc is removed.
bind-interfaces
except-interface=lxcbr0
server=/lxc/10.0.3.1
```

- This configuration of the system wide dnsmasq excludes lxcbr0, which is going to be managed by the specific dnsmasq of LXC.
- It also sends the DNS queries ending in .lxc to lxcbr0 (10.0.3.1) in which the lxc dnsmasq is resolving the names.
- Then, we configure the lxc dnsmasq by editing /etc/default/lxc-net as follows.

DNS Resolution iii

```
# This file is auto-generated by lxc.postinst if it does not
# exist.  Customizations will not be overridden.
# Leave USE_LXC_BRIDGE as "true" if you want to use lxcbr0 for your
# containers.  Set to "false" if you'll use virbr0 or another existing
# bridge, or mavlan to your host's NIC.
USE_LXC_BRIDGE="true"
# If you change the LXC_BRIDGE to something other than lxcbr0, then
# you will also need to update your /etc/lxc/default.conf as well as the
# configuration (/var/lib/lxc/<container>/config) for any containers
# already created using the default config to reflect the new bridge
# name.
# If you have the dnsmasq daemon installed, you'll also have to update
# /etc/dnsmasq.d/lxc and restart the system wide dnsmasq daemon.
LXC_BRIDGE="lxcbr0"
LXC_ADDR="10.0.3.1"
LXC_NETMASK="255.255.255.0"
LXC_NETWORK="10.0.3.0/24"
LXC_DHCP_RANGE="10.0.3.2,10.0.3.254"
LXC_DHCP_MAX="253"
# Uncomment the next line if you'd like to use a conf-file for the lxcbr0
# dnsmasq.  For instance, you can use 'dhcp-host=mail1,10.0.3.100' to have
# container 'mail1' always get ip address 10.0.3.100.
#LXC_DHCP_CONFILE=/etc/lxc/dnsmasq.conf
# Uncomment the next line if you want lxcbr0's dnsmasq to resolve the .lxc
# domain.  You can then add "server=/lxc/10.0.3.1' (or your actual )
# to /etc/dnsmasq.conf, after which 'container1.lxc' will resolve on your
# host.
# By j3o to activate DNSmasq with container_name.lxc
```

DNS Resolution iv

```
LXC_DOMAIN="lxc"
```

- Finally, we restart the system-wide dnsmasq and the lxc network (which includes the lxc dnsmasq):

```
hyp# systemctl restart dnsmasq.service
hyp# systemctl restart lxc-net.service
```

- Now, we can access our containers by name from the host:

```
hyp# ssh mycontainer.lxc
```

Basic Data Link Layer i

- Let's consider that we have created a container called mycontainer.
- Then, the configuration of the container is in the file **/var/lib/lxc/mycontainer/config**.
- The main L2 parameters that can be configured:

```
lxc.utsname = mycontainer
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = lxcbr0
lxc.network.hwaddr = 4a:49:43:49:79:bf
lxc.network.name = lxcnet0
lxc.network.veth.pair = vethmycontainer
```

- The configuration file specifies that:
  - We are going to use a veth interface.
  - Connected to a bridge in the host called `lxcbr0`.
  - The veth interface must be activated (up).
  - The hardware address that we will use is 00:16:3e:42:1d:a7.
  - The name of network device inside the container defaults to `eth0`.

Basic Data Link Layer ii

  - But in this case we set another name: `lxcnet0`.
  - In addition, we can select the interface name in the host.
  - In this case `vethmycontainer` (otherwise, by default LXC selects a random name like `veth3VcOob`).

- The default configuration for the previous parameters are in the file /etc/lxc/default.conf:

```
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = lxcbr0
lxc.network.hwaddr = 00:16:3e:xx:xx:xx
```

- Finally, after making any changes you will have to restart the container:

```
lxc-stop -n mycontainer
lxc-start -n mycontainer -d
```

Adding More Interfaces

- You can add more interfaces to a container by simply adding more L2 parameters:

```
lxc.utsname = mycontainer
# First Interface
lxc.network.type = veth
lxc.network.flags = up
# Second Interface
lxc.network.type = veth
lxc.network.flags = up
```

- By default, the second interface is called `eth1` and so on.

Using Open vSwitch

- If you want to configure a switch other than **brctl**, you can use a script to connect the interface to the bridge:

```
lxc.network.type = veth
lxc.network.flags = up
# lxc.network.link = br1 # Has to be disabled
lxc.network.script.up = /etc/lxc/ovs-br1.sh
lxc.network.name = eth1
```

- In this case, we create a new interface **eth1** connected to an OVS switch called **br1**.
- /etc/lxc/ovs-br1.sh is called when the container is started.
- This script could have a configuration like the following:

```
#!/bin/bash
BRIDGE="br1"
ovs-vsctl --may-exist add-br $BRIDGE
ovs-vsctl --if-exist  del-port $BRIDGE $5
ovs-vsctl --may-exist add-port $BRIDGE $5
```

IP Configuration

- As mentioned, by default the container obtains the IP addresses from a DHCP server but we can also configure them:

```
# The ip may be set to 0.0.0.0/24 (not assigned)
# or skip these lines if you like to use a dhcp
# client inside the container
lxc.network.ipv4 = 1.2.3.5/24
lxc.network.ipv6 = 2001:db8:fab::1
```

- If you do not want to assign an IP address use 0.0.0.0/24 and if you are using a DHCP server, comment these lines.

Types of Interfaces i

- We have several types of network virtualizations can be used with LXC containers.
- These are defined in the configuration file with the parameter lxc.network.type.
- Each time a lxc.network.type field is found a new round of network configuration begins.
- In this way, several network virtualization types can be specified for the same container, as well as assigning several network interfaces for one container.

Types of Interfaces ii

- The different virtualization types can be:
  1. **`none`**: will cause the container to share the host's network namespace.
     - This means the host network devices are usable in the container.
     - Both the container and host will have the same INIT system, in other words, a '`halt`' in the container will shut down the host.
  2. **`empty`**: will create only the loopback interface.
  3. **`vlan`**: a vlan interface is linked with the interface specified by the lxc.network.link and assigned to the container. The vlan identifier is specified with the option lxc.network.vlan.id.
  4. **`macvlan`**: a macvlan interface is used.
     - A macvlan interface is linked with the interface specified by the lxc.network.link and assigned to the container.
     - lxc.network.macvlan.mode specifies the mode the macvlan will use to communicate between different macvlan on the same upper device.

Types of Interfaces iii

5. **veth**: uses veth.
   - A virtual ethernet pair device is created with one side assigned to the container and the other side attached to a bridge specified by the lxc.network.link option.
   - If the bridge is not specified, then the veth pair device will be created but not attached to any bridge.
   - Otherwise, the bridge has to be created on the system before starting the container.
   - lxc won't handle any configuration outside of the container.
   - By default, lxc chooses a name for the network device belonging to the outside of the container.
   - But if you wish to handle this name yourselves, you can tell lxc to set a specific name with the lxc.network.veth.pair option (except for unprivileged containers where this option is ignored for security reasons).
6. **phys**: an already existing interface specified by the lxc.network.link is assigned to the container.

MAC VLAN Interace i

- The **mcvlan** (MAC VLAN) is a way to take a single network interface and create multiple virtual network interfaces with different MAC addresses:
  - It is like creating a VLAN but not using the 802.1Q tag but the mac address.
  - Macvlan interfaces can be seen as subinterfaces of a main ethernet interface.
  - Each macvlan interface has its own MAC address (different from that of the main interface).
  - We can assign an IP address to a macvlan just like a normal interface.
  - I.e. it's possible to have multiple IP addresses, each with its own MAC address, on the same physical interface.
  - Apps can bind specifically to the IP address assigned to a macvlan interface.
- The main use of macvlan is in virtualization (containers and VMs).

MAC VLAN Interace ii

- The physical interface to which the macvlan is attached is often referred to as the **upper device** (sometimes also as the **lower device**).
- A macvlan interface can work in one of four modes, defined at creation time:
  1. **Private mode**:
     - A macvlan device never communicates with any other device on the same upper_dev (default).
     - Macvlans on the same upper device can not communicate.
     - Regardless of where the packets come from.
     - So even if inter-VM traffic is sent back by a hairpin switch or an IP router, the target macvlan is prevented from receiving it.
     - This mode is useful if we really want macvlan isolation.

MAC VLAN Interace iii

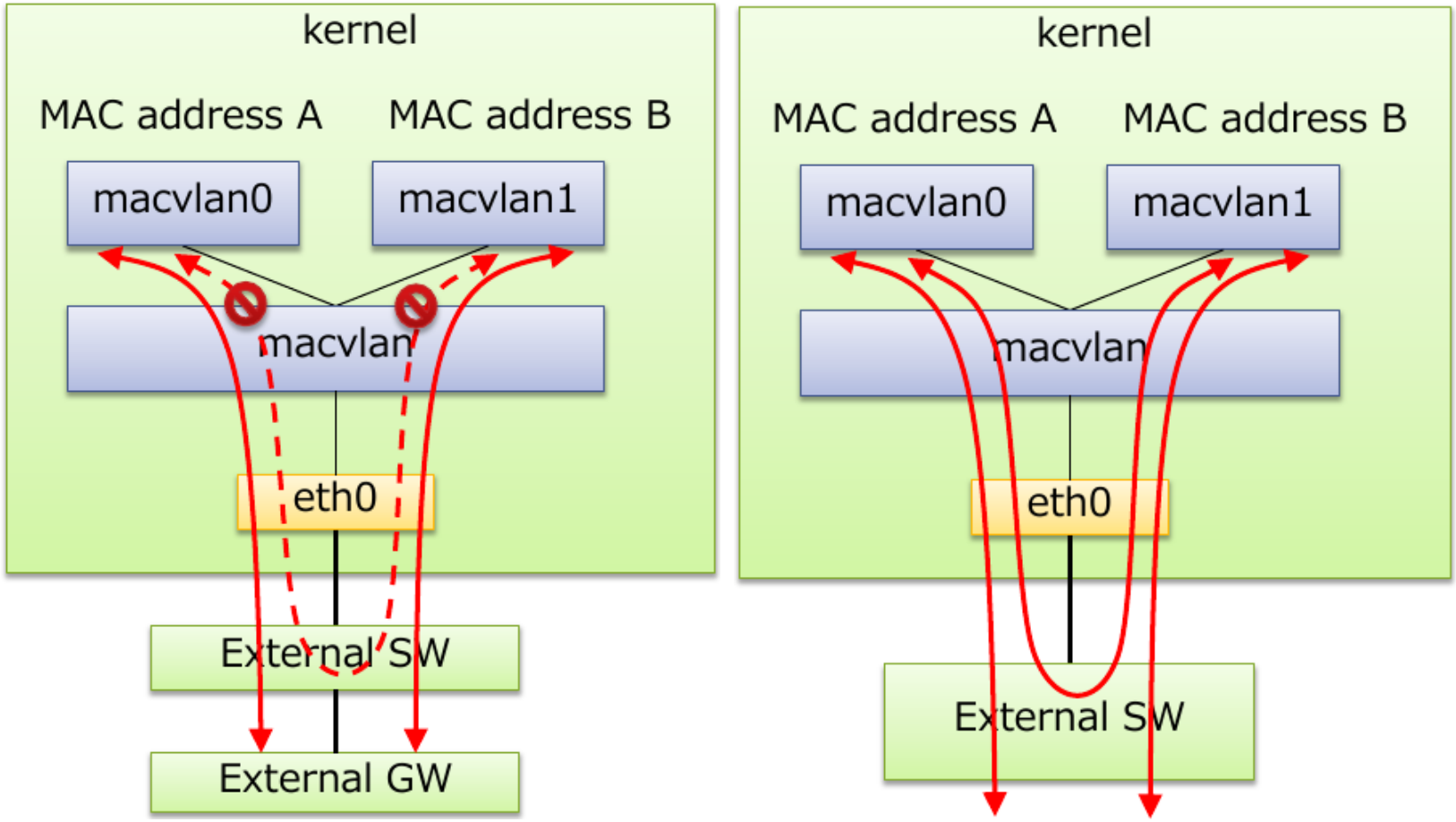
2. **VEPA (Virtual Ethernet Port Aggregator)**:
   - If the upper device receives data from a macvlan in VEPA mode, this data is always sent ”out” to the upstream switch or bridge.
   - Even if it’s destined for another macvlan in the same upper device.
   - When used in VMs or containers it makes possible to manage inter-VM traffic on a real external switch.
   - The adjacent bridge returns all frames where both source and destination are local to the macvlan port.
   - The bridge is said to be configured in ”hairpin mode” or ”reflective relay”.

MAC VLAN Interace iv

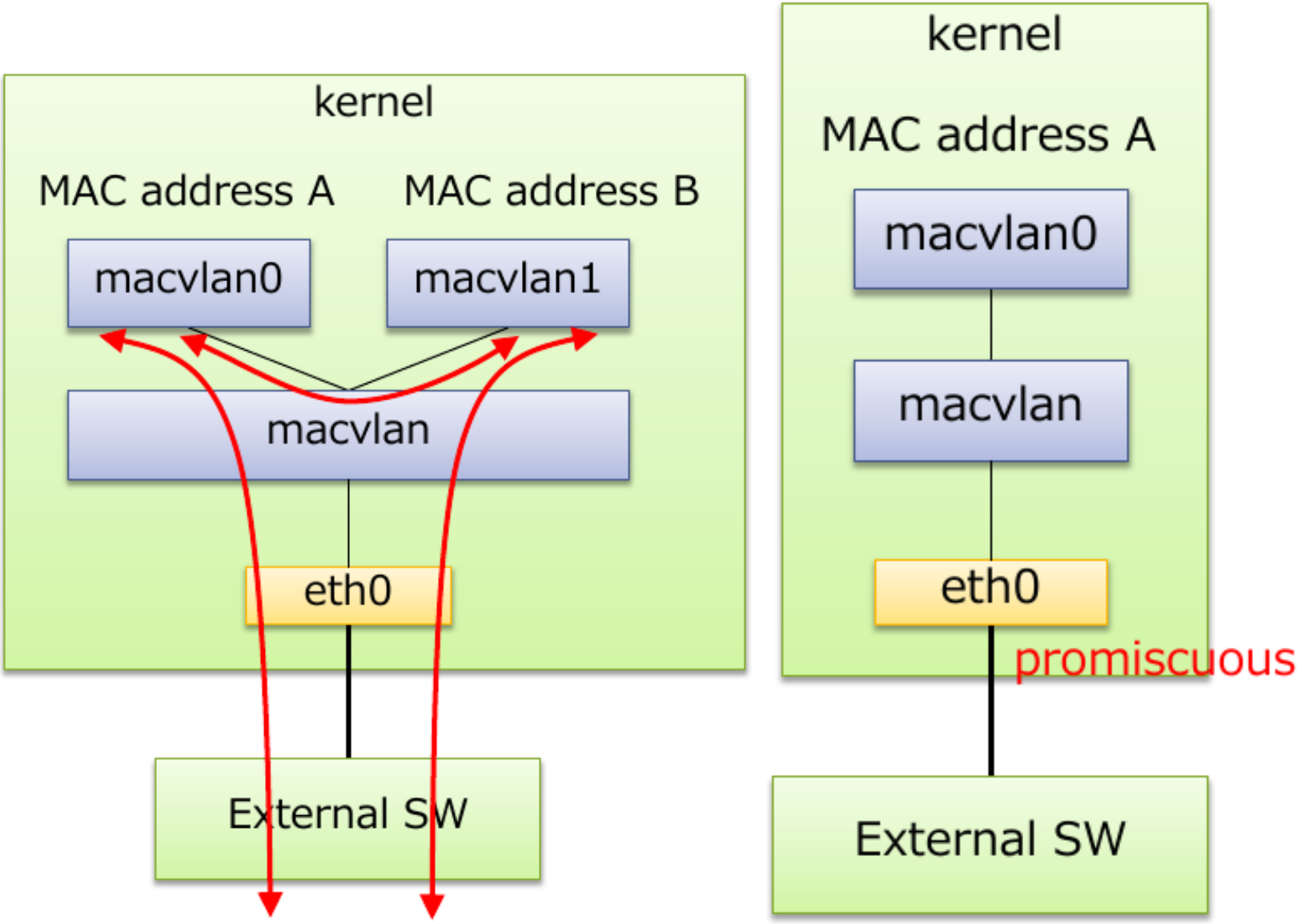
3. **Bridge mode**:
   - This mode works almost like a traditional bridge.
   - Data received on a macvlan in bridge mode and destined for another macvlan of the same upper device is sent directly to the target (target macvlan also in bridge mode).
   - Works well with non-hairpin switches.
   - Inter-VM traffic has better performance than VEPA mode, since the external round-trip is avoided.
   - Broadcast frames get flooded to all other bridge ports and to the external interface.
   - But when they come back from a reflective relay, we don't deliver them again.
   - Since we know all the MAC addresses, the macvlan bridge mode does not require learning or STP like a conventional bridge.

4. **Passthru mode**:
   - This mode was added later.
   - The vepa mode has the following limitations on the guest VM:
     - Cannot change/add a mac address.
     - Cannot create a vlan device.
     - Cannot enable promiscuous mode.
   - To address these limitations, passthru allows takeover of the underlying device and passing it to a guest.
   - Only one macvlan device can be in passthru mode.
   - It inherits the mac address from the underlying device.
   - It is set in promiscuous mode to receive and forward all the packets.

MAC VLAN Interace vi



MAC VLAN Interace vii



Notes About MACVLAN

- In both VEPA and Passthru modes:
  - Traffic destined to a different MAC VLAN on the same interface will transit the physical interface twice.
  - Once to egress the interface where it is switched.
  - Then, sent back and ingresses via the same interface.
  - This can affect available physical bandwidth.
  - Also restricts inter MAC VLAN traffic to the speed of the physical connection.
- In Private mode:
  - No node may communicate with each other.
  - This can help prevent discovery of other MAC VLANs.
  - May be useful in a multi-tenant environment in conjunction with a switch that sends all traffic to a router.
- In Bridge mode:
  - All traffic between MAC VLANs will be switched in memory.
  - This can lead to higher network speeds between interfaces than VEPA or Passthru due to memory typically being faster than network interfaces.

MACVLAN in Linux i

- In a Linux system you can add a macvlan interface with the command:

```
# ip link add link IF macvlan type macvlan \
mode [private|vepa|bridge|passthru]
```

- E.g. create a MAC VLAN interface and auto generate a MAC address:

```
# ip li add link eth0 mac0 type macvlan
```

- This will create a new interface called `mac0@eth0`.
- You can create another one:

```
# ip li add link eth0 type macvlan
```

- If you do not specify the name the inferface is called `macvlan0@eth0`.

MACVLAN in Linux ii

- Note that to bring the interface UP:
  - The virtual macvlan link must be UP.
  - Also the main interface must be UP.
- You do not need to specify the full `mac0@eth0` to make changes to the interface, just `mac0`:

```
# ifconfig -a
....
mac0      Link encap:Ethernet  HWaddr 3a:0a:f3:60:93:4f
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

macvlan0  Link encap:Ethernet  HWaddr 0a:55:1e:35:bf:3d
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
....
```

MACVLAN in Linux iii

- If you need to create a MAC VLAN with a specific mac address use the form:

```
# ip li add link eth0 mac1 \
address 56:61:4f:7c:77:db type macvlan
```

- And how to show extended information including the mode (default is VEPA):

```
# ip -d link show
.....
x: macvlan0@eth0: <BROADCAST,MULTICAST> mtu 1500
                  qdisc noop state DOWN mode DEFAULT
                  group default qlen 1
    link/ether 0a:55:1e:35:bf:3d brd ff:ff:ff:ff:ff:ff
    promiscuity 0 macvlan  mode vepa addrgenmode eui64
```

MACVLAN in Linux iv

- Assuming `eth0` is a port of `br0`, hairpin mode can be anabled by doing:

```
# echo 1 > /sys/class/net/br0/brif/eth0/hairpin_mode
```

- Or using a recent version of brctl:

```
# brctl hairpin br0 eth0 on
```

- Or even better, using the bridge program that comes with recent versions of iproute2:

```
# bridge link set dev eth0 hairpin on
```

Outline

Container Migration

- To statically move a container from a host to another host:
  1. We stop the container
  2. We make a tar on the current host:

```
# tar --numeric-owner -czvf mycontainer1.tgz \
/var/lib/lxc/mycontainer1
```

- We copy the tar on the remote host:

```
# cd /var/lib/lxc/
# tar --numeric-owner -xzvf mycontainer1.tgz
```

Outline

Configuration Directories

Directory	Description
/var/lib/lxc	Store containers (filesystem and config).
/var/lib/lxcsnaps	Store container snapshots (filesystem and config).
/var/cache/lxc	Store cached downloaded data.
/etc/lxc	Store default.conf which is the file with the default LXC parameters.
/etc/init	Store lxc-net.conf which is the default upstart networking configuration

Outline

Unprivileged containers i

- Unprivileged containers are run without root access:
  - Are safer to run since they run in the context of a non-sudo user.
  - Handy when share a server with other users.
  - Or when you do not want to provide your containers full access to your underlying system for security reasons.
- To run them, we must install systemd-services and uidmap:

```
# apt-get install systemd-services uidmap
```

- Next, we need to create some configuration at user's home.
- Equivalence with system config is:

System Configuration Directory	User's Configuration Directory
/var/lib/lxc	$HOME/.local/share/lxc
/var/lib/lxcsnaps	$HOME/.local/share/lxcsnaps
/var/cache/lxc	$HOME/.local/cache/lxc
/etc/lxc	$HOME/.config/lxc

Unprivileged containers ii

- At a minimum create:

```
hypervisor@user1$ chmod +x $HOME
hypervisor@user1$ mkdir .config/lxc .local/share/lxc
hypervisor@user1$ mkdir .local/cache/lxc
```

- The first two to store containers and the third for the downloaded container templates cache.

Unprivileged containers iii

- When a new user is created in Ubuntu:
  - Automatically assigned a range of 65536 user and group ids.
  - Start at 100000 to avoid conflicts with system users.
- If you look at /etc/subuid and /etc/subgid:

```
user1:100000:65536
user2:165536:65536
```

- This indicates that:
  - user1: user and group ids from 100000 to 165535
  - user2: user and group ids from 165536 to 231072.
- If you need it, the commands to allocate additional uids and gids to your username are the following:

```
# usermod --add-subuids 100000-165536 user1
# usermod --add-subgids 100000-165536 user1
```

Unprivileged containers iv

- User/group ids are mapped in each unprivileged container (will start at 0).
- Now in .config/lxc create a default.conf file like (each user must specify its own allocated range):

```
lxc.network.type = veth
lxc.network.link = lxcbr0
lxc.network.flags = up
lxc.network.hwaddr = 00:16:3e:xx:xx:xx
lxc.id_map = u 0 100000 65536
lxc.id_map = g 0 100000 65536
```

- Network:
  - We need to allow the user to hook into the container network through the bridge lxcbr0.
  - We need to configure the file /etc/lxc/lxc-usernet for unprivileged containers.
  - A configuration line in this file can be the following:

```
user1 veth lxcbr0 5
```

Unprivileged containers v

  - In this example, we let user1 be able to create up to 5 veth interfaces in the bridge `lxcbr0`.

- Finally, to create unprivileged containers you need to use the 'download' template type.
- These container OS templates are designed to support unprivileged containers:

```
hypervisor~user1$ lxc-create -t download \
-n mycontainer1 -- -d ubuntu -r trusty -a amd64
```

- This should create the mycontainer1 container in your user's .local/share/lxc folder.
- Now start the container and get a console:

```
hypervisor~user1$ lxc-start -n mycontainer1 -d
hypervisor~user1$ lxc-console -n mycontainer1
```