Introduction

This document describes the mechanism by which a Northstar container can be launched in a Firecracker VM (called *Northcracker* in this document). It is assumed the reader is familiar with Northstar and with virtual machine fundamentals.

Goal

The goal is to make it completely transparent to the creator of the container whether the container is to be run "normally" (in a chrooted, jailed process on the host) or in a VM.

Factors that guide the decision to use a VM include:

Pro	Con
Security and isolation: it is typically much harder for a compromised application to wreak havoc on a host when run in a VM.	*Startup time*: it can take over a second for an application running in a VM to a begin execution, compared to a few hundred milliseconds when run natively.
Seccomp: it is easier to develop a seccomp list for a VM because the number of system calls performed by the hypervisor is well known. Most likely, the list can be taken from the Firecracker code base.	*Performance*: while performance in a VM is good, it will always be slightly slower than native.
	Environment: dynamic linking with host libraries is not available in the VM. The binaries must either be MUSL (alpine Linux) dynamic, or statically linked.

When executing, the application should not notice any difference between executing natively in a Northstar container or in a VM.

Procedure

The basic steps to a running a Northcracker VM are as follows:

- Setup network namespace
- Create config files for Firecracker and the VM
- Setup bind mounts for target container
- Exec VM launcher (instead of target application)
- Setup VM communication `vsocks`
- Launch Firecracker using kernel and *initrd* from a bind-mounted resource container
- Configure VM networking and mounts
- Launch application from *virtio* device mounted from host

setup phase

launch phase

execution phase

All of the required resources to launch a VM are encapsulated in a resource container. This resource container is used during setup (before container launch) and when the container is launched.

Network namespaces

A network namespace is required when running in a VM. This is to simplify the network configuration, and to prevent any chance of a network change in the container from affecting the host.

The approach to configuration was to make it as simple as possible to configure for each container.

> *TL;DR : To configure a network namespace, specify a bridge address in* `north.toml` *and a subnet number in the container manifest.*

The network bridge and a class B IPv4 address must be specified in the `north.toml` file. Each network namespace is assigned to a subnet address range; the network namespace itself is given a fixed address within that subnet, as is the VM.

The bridge is created when the north runtime starts and is torn down when the runtime exits.

For example, if the `north.toml` file specified:

```
[bridge]
  enabled = true
  ipv4_slash16 = "172.30.0.0"
```

then a bridge would be created in the root namespace with the name `nstar0` with an IPv4 address of `172.30.0.1` . The name of the bridge can not be changed. The bridge is given a unique MAC address. This bridge IP address is used as the default gateway for all namespaces and VMs.

The root namespace will have an `iptables` rules added to enable *masquerading* of all addresses within the `ipv4_slash16` address range. For example given above, the basic rule would be

```
iptables -t nat -A POSTROUTING -s 172.30.0.0/16 ! -o nstar0 -j MASQUERADE
```

The bridge and masquerade rules are created when the north runtime starts. When the north runtime exits, the bridge and rules are removed.

To enable network namespaces, the container **manifest** must contain a `netconf` section:

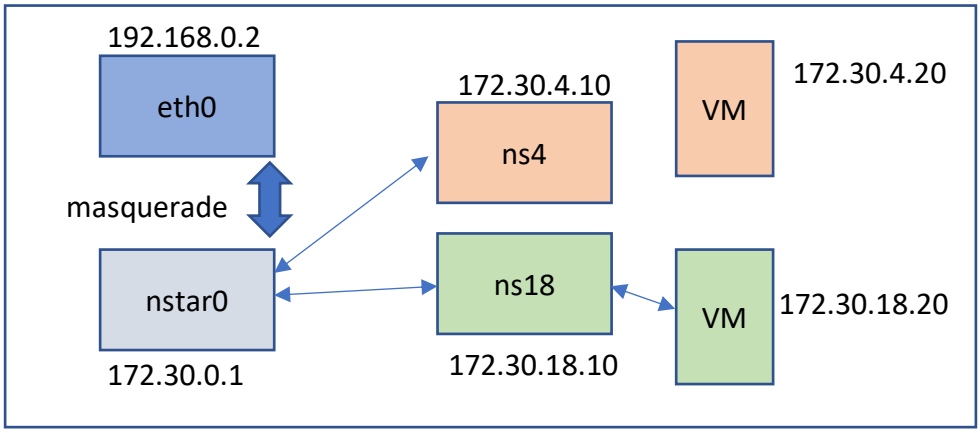
```
netconf:
   enable_namespace: true
   namespace_uniq_subnet: 4
```

The `namespace_uniq_subnet` subnet must be unique among all containers; it is the responsibility of the system integrator to ensure that there are no duplicates among all containers. If a duplicate subnet ID exists in the registry, the container will fail to launch.

The namespace (and VM, if configured) are assigned fixed addresses within the subnet.

Using the example above would result in a container namespace with an address of `172.30.4.10` . If a VM is configured, it will have the address of `172.30.4.20` . The network gateway for routing is based on the address specified in the `north.toml` file and in this example will be `172.30.0.1` .

The namespace will have the name of `ns<subnet>`; in the above example the namespace would be named `ns4`. The namespace will have a unique MAC address. If a second namespace is configured with a `namespace_uniq_subnet` of 18, the result would be as in the following diagram:



Namespaces are created dynamically when the container is set up; once the container is launched, all references to the namespace are removed so that when the container exits, the namespace is automatically removed. This is akin to unlinking a file that is in use by a process; when the process exits, the last reference goes away and the file is removed.

Android

On Android, if DNS is required in the namespace, then the entire `/dev` directory tree must be bind mounted into the container. This is controlled by the following in the manifest:

```
mounts:
  /dev: full
```

The reason is that Android uses `/dev/socket/dnsproxyd` for DNS resolution; if a skeleton `/dev` exists, then DNS queries will fail because no socket connection is available to the resolver.

To access the network stack in a container on Android, certain permissions are required. The process must be in the *inet* group (3003). Without group permissions, the container must have the `CAP_NET_RAW` *capability* to do a ping.

KVM

KVM requires a *tap* device to present the network devices to the VM. Using network namespaces allows the creation of a *macvtap* network device instead of a *tun/tap* device. This ties it to the network namespace, and means that when the process exits, the network device is automatically torn down. The *macvtap* device also does not require a bridge; it can

be tied directly to another network device. This reduces the number of devices that must be created during setup. See the `ip-link(8)` man page for more information.

In addition to the network devices that are created in the context of the network namespace, a device node must also be created with the appropriate major and minor numbers to allow communication with the kernel using the KVM ioctls.

Northstar processes run inside of mount namespaces. Mount namespaces use a tag to determine what should be visible to the running process. When adding a network namespace, this requires that the `/sysfs` directory hierarchy be remounted so that the correct information is visible inside the namespace. The `ip` utility takes care of this automatically when executing a shell inside a namespace; when manually adding a namespace, this step must be performed manually.

The psuedo-code is as follows:

```
/*
 * We need a new mount namespace for manipulating /sys
 */
 unshare(CLONE_NEWNS);

/*
 * See netns_switch() in iproute2-5.9.0/lib/namespace.c
 *
 * This is crucial to avoid screwing up the parent host
 * Don't let any mounts propagate back to the parent
 */
 mount("", "/", "none", MS_SLAVE | MS_REC, NULL)

/*
 * A remount will only change the flags, and we need to get the
 * kernel to pick up the namespace tags associated with
 * /sys/class/net, so we do a full mount.
 */
umount2("/sys", MNT_DETACH);

/*
 * Now finally do the real mount
 */
mount(nsname, "/sys", "sysfs", 0, NULL);
```

After `/sysfs` has been remounted, the major and minor numbers for the device node can be determined from the `ueventd` file, and the device entry created in `/dev` for KVM and Firecracker.

Firecracker patches

Firecracker does not yet officially support macvtap. A PR is open and under discussion (https://github.com/firecracker-microvm/firecracker/pull/2217). The fix is to allow full path names for the device entry; the issue is how snapshots are handled. However, since snapshot handling is not a concern for Northstar, the patch as it exists works just fine.

Setup Phase

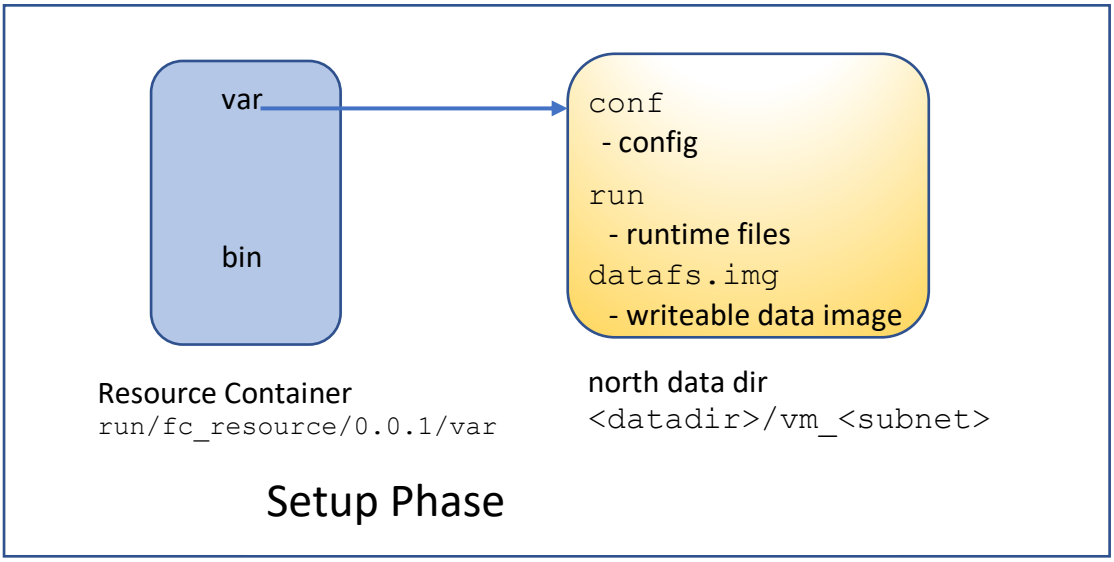
There are two configuration files that must be created to launch a VM. The first is the Firecracker config file that specifies the location of various files (such as the kernel and initrd image) as well as the location of the control sockets and the path to the KVM tap device.

The second is the VM configuration file that contains the application settings from the container manifest and the network block device configuration. These two config files are written during the setup phase, before the target container is launched by minijail.

Resource container

A resource container (named *fc_resource*) contains all the files required to launch a VM. This includes the Firecracker binary, kernel for the VM, *initrd* for the VM, and template configuration files.

During setup (before the container is launched by minijail), the resource container has already been mounted by Northstar (all containers are mounted at `<rundir>/<name>/<version>` at runtime start). The config files in the resource container are used as a template, and modified versions are written to the container's `<data>` directory under `<datadir>`**`vm_<`**`subnet_index>` . This directory will be mounted into the target container so that Firecracker can reference the updated config files.



The target container must be able to reference both the resource container and the newly created configuration files. There are two bind mounts that accomplish this.

> **Workaround**: For now, the bind mounts are at `/usr/local` in the target container. This means that the target container must have these directories in its image. Alternatively, the bind mounts could be something in a `tmp` directory, since the mounts are not required by the container application, but only by the mechanism to launch the container.

The rationale for this approach is to keep in line with the goal of transparency; there should be minimal differences in the configuration of a container that runs natively, and one that runs in a VM.

Since the manifest has a tag that specifies running in a VM, one approach would be to create a uniquely named mount directory during container creation if the container is to be run in a VM. This is TBD.

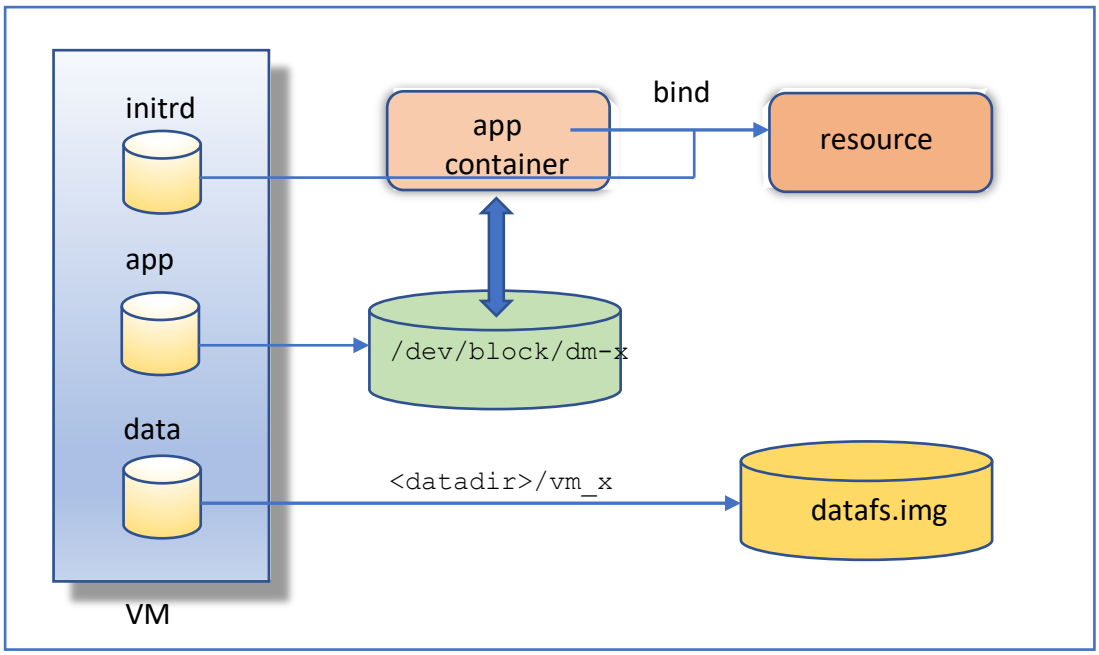
The first bind mount is of the resource container into the target; this is a read-only mount (since the container is read-only by default). During the launch phase, the Firecracker binary, kernel, and *initrd* all come from the resource container accessed through this bind mount.

The second bind mount is writable, and contains the Firecracker log files and all output from the container. The output from application in the VM is redirected to a file. This file can then be dumped into the log. There is also a 256MB filesystem image that is presented to the VM as a writeable `/data` partition. Since this file lives on the host, it is persistent and can be accessed after the VM has exited.

VM Block Devices

Firecracker uses *`virtio`* devices for the block device interface. This means the source must be a file (such as a filesystem image) or a block device. It is not possible to bind-mount from host into VM. For sharing of data, one future path might be `virtiofs`, but that is not currently being considered by Firecracker as they do not need it. Sharing data between host and VM also goes against the design principles of Firecracker, so it is highly unlikely that filesystem sharing will ever be implemented in Firecracker.

The Firecracker config file specifies the source for each of the *virtio* devices presented to the host. The first must be the *initrd* image and is accessible in the VM as `/dev/vda` . The second device is the writeable filesystem image (as `/dev/vdb`). Lastly, the block device for the target container is specified so that the VM can mount the container (for example `/dev/block/dm-0` on the host becomes `/dev/vdc` in the VM). In this manner are all the files in the container accessible on the host. This mount is read-only.



Execution Target

During the setup phase, the target of minijail's `exceve()` must be altered so that instead of launching the application, it executes a launch process. This launch process will create the necessary environment for executing Firecracker and launching the VM. This requires updating the `/init` string and the `argv[]` vector.

Device Permissions

Since north can run with a `uid/gid` other than root, it is important that the devices be accessible by random `uid/gid`. Since the `uid/gid` is currently only specified in the `north.toml`, the workaround for now is to `chmod 666` the necessary device files so that they are accessible in the target container. This workaround also applies to the device nodes for kvm and the macvtap.

Launch Phase

Once the launch process begins running in the context of the sandboxed container, it uses the binaries from the resource container to start. The path is fixed `(/usr/local`, as discussed above) for each VM, since the resource container is bind-mounted into each container.

Firecracker uses *vsocks* to communicate with a VM. A vsock appears on the host as a unix domain socket but the other end is intercepted by the hypervisor and made available to the VM. Each socket is referenced by a port number. In the launch phase, there are three vsocks that are created via `socat`. `Socat` was chosen for ease of use, and the binary is in the resource container.

- port 0: configuration (think: *stdin*)
- port 1: logging (*stdout*)
- port 2: status (*stderr*)

Prior to starting Firecracker, the launch process must start the `socat` processes. The *configuration* `socat` waits for a connection from the VM and uploads the VM configuration file when connection is established. The *logging* `socat` process takes all output from the VM and writes it to a file. The *status* `socat` process receives the exit code of the target application in the VM, since the exit code is not available via any other mechanism.

The results of all `socat` processes are available in the writeable bind-mounted data directory (`<datadir>/vm_<subnet>/run/fclog`)

Once the `socat` processes have been started, then Firecracker is invoked using the configuration file that was written during the setup phase. Firecracker will exit when the VM exits.

Execution Phase

When the VM begins executing, a simple `init` process begins executing. The `init` process first retrieves the VM configuration file from the host over a vsock. In the VM, it uses `nc-vsock` to read the data from the vsock.

There are two parts to configuring the VM; setting up the network based on the parameters in the config file and mounting the devices. It typically takes from 750ms to 1.5 seconds to configure the network and mount the two devices in the config file (`/data` and `/app`) on a two core A57 B1 sample board.

After the VM has been initialized, the `init` process then invokes a wrapper for the application.

> **Workaround**: for now, the path to the application binary is modified during the setup phase to include the `/app` path. However, this means that the app container must be set up to execute out of `/app`.
>
> The environment in a VM is different than on the host, and no bind mounts are available. This means the list of mounts in the container manifest are not applicable, as they pertain to the host.
>
> It still might be better to set up a *chroot* environment so that the execution directory is the same between VM and container. This is TBD, and requires real use cases.

The most important role of the application wrapper is to capture all output and redirect it over a vsock to the host. This is because otherwise it would go to stdout which in the VM is a serial console, and that is painfully slow.

The application wrapper gets the environment variables from the VM config file and exports them. It then invokes the application and waits for it to exit. It takes the exit code and sends it via the status vsock to the host.

Environment

The VM environment is currently an Alpine Linux image, with a minimal number of packages (see https://pkgs.alpinelinux.org/packages) . Currently the installed list is:

Package	Purpose
apk add `openssh`	currently unused; might make debug easier
apk add `util-linux`	base utilities
apk add `strace`	very helpful for debug
apk add `jq`	parsing json config file
apk add `mosquitto-clients`	testing

For more information on the initrd and the steps to create it, there is a README in the resource container at *root/var/README.txt*

Since bind mounts to the host are not available, it is not possible to share dynamic libraries between host and guest. This means that the application must be either statically linked or completely self-sufficient (supplying all required libraries).

It is also worth noting that the Alpine packages use MUSL, not glibc .

Testing

To verify operation, there are a few test containers, based on the hello example. They are used to verify correct network namespace operation by performing the following:

- ping 8.8.8.8
- ping `google.com`
- *mqtt* publish to broker listening on gateway address.
- verify that network namespace is different than host network namespace

With the introduction of groups and capabilities, the test container also verifies that it has `CAP_NET_RAW` (required for ping if running without groups), and has the correct group memberships (as specified in the manifest file).

When running in a VM, the network tests are the same. However, the groups and capabilities sections are skipped as they only work on native containers.

Summary

The goal of having the container work both in a VM and as a "normal" Northstar container while laudable, is not practical. The developer should, and must, know the environment that the application will be running in, from the simplest things like what libraries to link with to more complex topics such as what are the required system interactions. There is a big difference between how a Mindroid environment on a 2-core A57 appears to an application and how a VM looks to an application.

The choice of what environment to target (native Northstar container or VM) will depend on the level of system interaction required, the resiliency to boot time latencies, and the desire for the heightened security provided by a VM.

Open Items

This is just the beginning .....

Issue	Comment
Dump output from VM into logger	Currently just in a file in `/data` that is overwritten on each invocation
Persist writeable data file	Currently overwritten on each invocation.
Hard-coded mount points in container	Currently `/usr/local` ; should be transparent to container.
Add tag in manifest for resource container	Currently hard-coded name
How handle permissions on devices	Currently `chmod 0666`
Do chroot in VM ?	Execution environment should be orthogonal to running as native container

Oh, and the C to Rust linkage is beyond awful.

Firecracker config file template

The items in Capital Letters are replaced at runtime.

```
{
  "boot-source": {
    "kernel_image_path": "/usr/local/share/var/bzImage.guest",
    "boot_args": "console=ttyS0 quiet init=/init reboot=k panic=1 pci=off"
  },
  "drives": [
    {
      "drive_id": "rootfs",
      "path_on_host": "/usr/local/share/var/initrd.guest",
      "is_root_device": true,
      "is_read_only": true
    },
    {
      "drive_id": "data",
      "path_on_host": "/usr/local/data/datafs.img",
      "is_root_device": false,
      "is_read_only": false,
      "want_flush": true
    },
    {
      "drive_id": "app",
      "path_on_host": "APP_BLKDEV",
      "is_root_device": false,
      "is_read_only": true,
      "want_flush": false
    }
  ],
  "network-interfaces": [
    {
      "iface_id": "eth0",
      "guest_mac": "TAP_MACADDR",
      "host_dev_name": "TAP_DEVNAME"
    }
  ],
  "machine-config": {
    "vcpu_count": 2,
    "mem_size_mib": 512,
    "ht_enabled": false
  },
  "metrics": {
    "metrics_path" : "/usr/local/data/run/fcmetrics"
  },
  "vsock": {
    "vsock_id": "1",
    "guest_cid": 3,
    "uds_path": "/usr/local/data/run/fcvsock.sock"
  }
}
```

VM config file template

```
{
  "netconf": {
        "ipaddr": "IPADDR",
        "gateway": "GATEWAY",
        "cidr": "CIDR"
  },
  "mounts": [
    {
        "dev":  "/dev/vda",
        "mountpoint": "/",
        "flags": "-r"
    },
    {
        "dev":  "/dev/vdb",
        "mountpoint": "/data",
        "flags":  ""
    },
    {
        "dev":  "/dev/vdc",
        "mountpoint": "/app",
        "flags":  "-r"
    }
  ],
  "app" : {
        "init": "INIT",
        "args": "ARGS",
        "env": "ENV"
    }
}
```