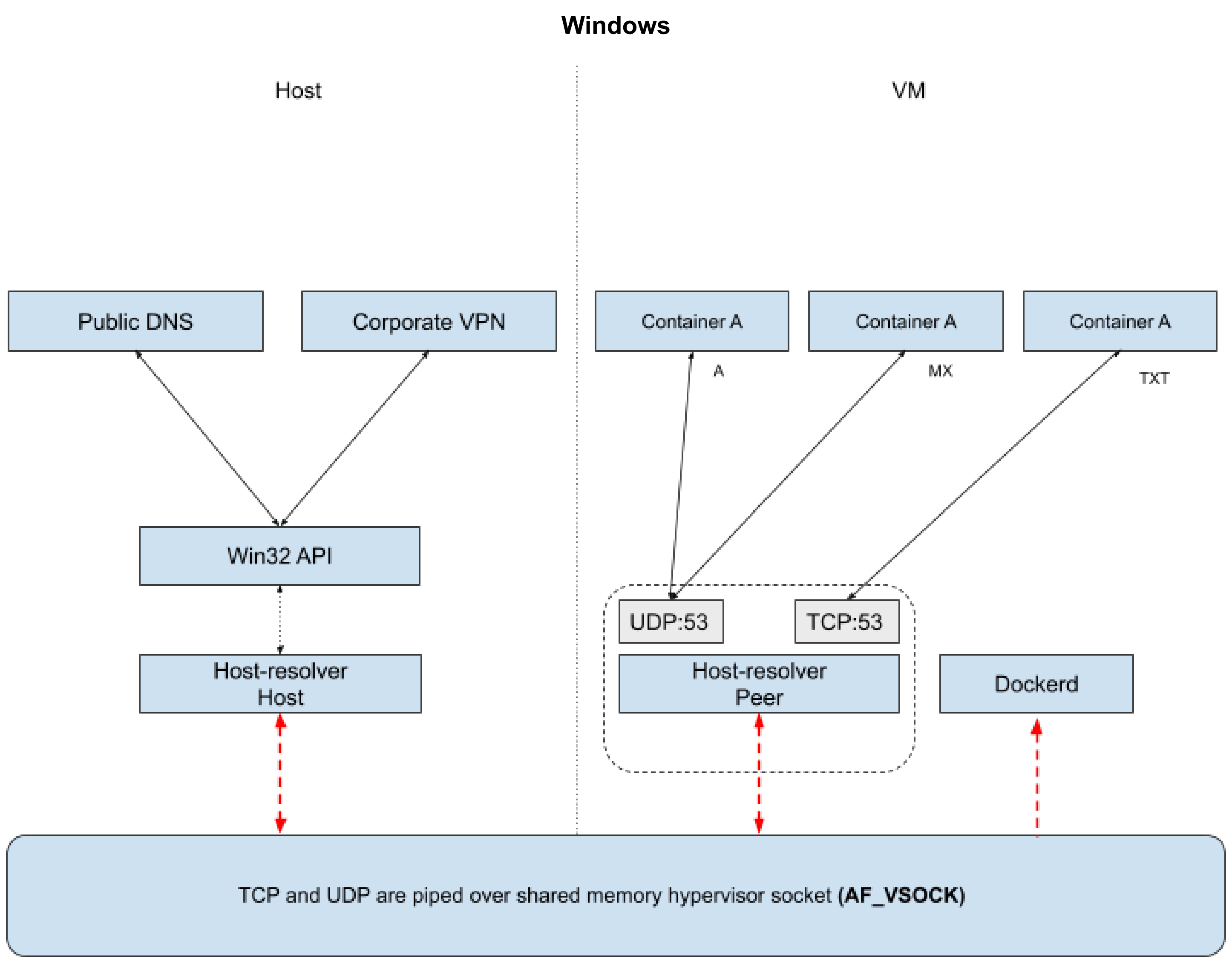
DNS

Our current architecture and how DNS lookups are handled while a split VPN tunnel is being configured are demonstrated below. This design is currently only available through a beta feature when experimentalHostResolver is set to enable. Currently, all DNS lookups inside the WSL distro VM are done through a host-resolver peer process that listens on both TCP and UDP port 53. The host-resolver peer process is an intermediary stub resolver for forwarding all the requests to its upstream stub server host-resolver host process that runs on the host machine.

The communication between these two processes takes place over a low-level transport shared memory AF_VSOCK which is one of many variations of the virtual socket address family. Containers and all other processes that require DNS lookup forward their DNS resource record request to the peer process in the VM either via TCP or UDP.

Host-resolver peer process then forwards the payload over the AF_VSOCK to the host-resolver host process and waits for the response from the host. Once the host-resolver host process receives the DNS question, it will then use the win32 API to retrieve the response. At this point, the request is handed over to the operating system's glibc which in turn uses the standard calls through getaddrinfo(windows) and gethostbyname(windows), etc to determine the resolutions using appropriate interfaces. This is how the host-resolver host process can distinguish name lookups that are destined for VPN vs the ones that are destined for regular internet.

The diagram below demonstrates our current architecture on the Windows platform.



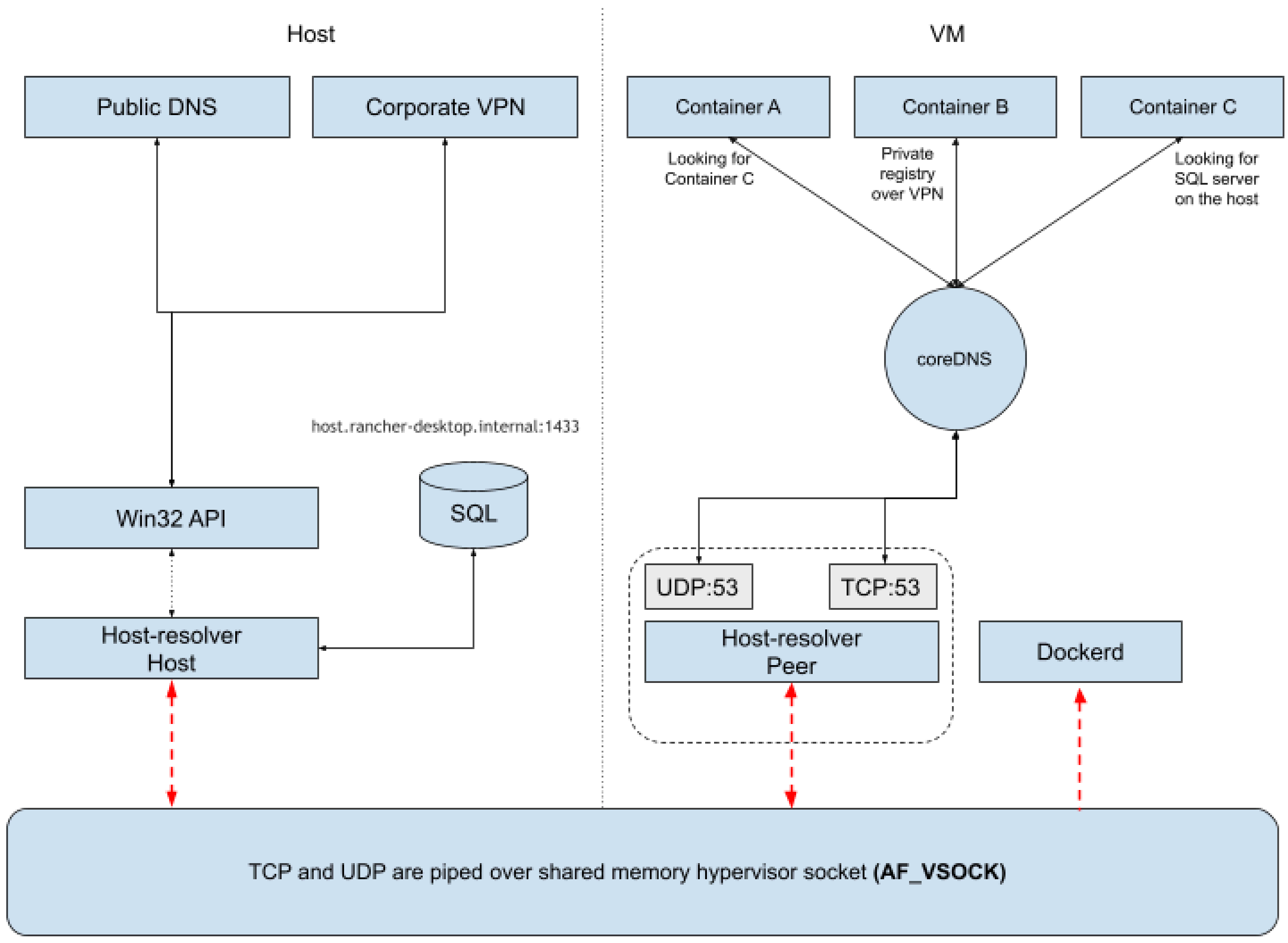
Something that can be added to this architecture in the near future is to run an additional DNS server in the WSL VM to handle container to container networking and name lookups. CoreDNS is a CNCF project and can be leveraged as an efficient forwarder and immediate resolver within the container namespace(s).

In the diagram below you can see that containers no longer talk to the host-resolver peer process directly, but rather their immediate DNS lookup takes place at the coreDNS. This design can be beneficial since it allows the containers to reference each other by using their name as opposed to the IP address. This can be helpful when a container is restarted and changes its IP address.

If a name lookup is destined for one of the neighboring containers, coreDNS will immediately resolve the lookup without having to talk to its upstream host-resolver peer process. However, if a lookup is destined for something beyond coreDNS's knowledge then recursive DNS lookup will take place with host-resolver peer being the first node and coreDNS will act as a forwarder to forward the requests to its peer (host-resolver peer process).

You can see in the diagram below the architecture with coreDNS as an immediate forwarder.

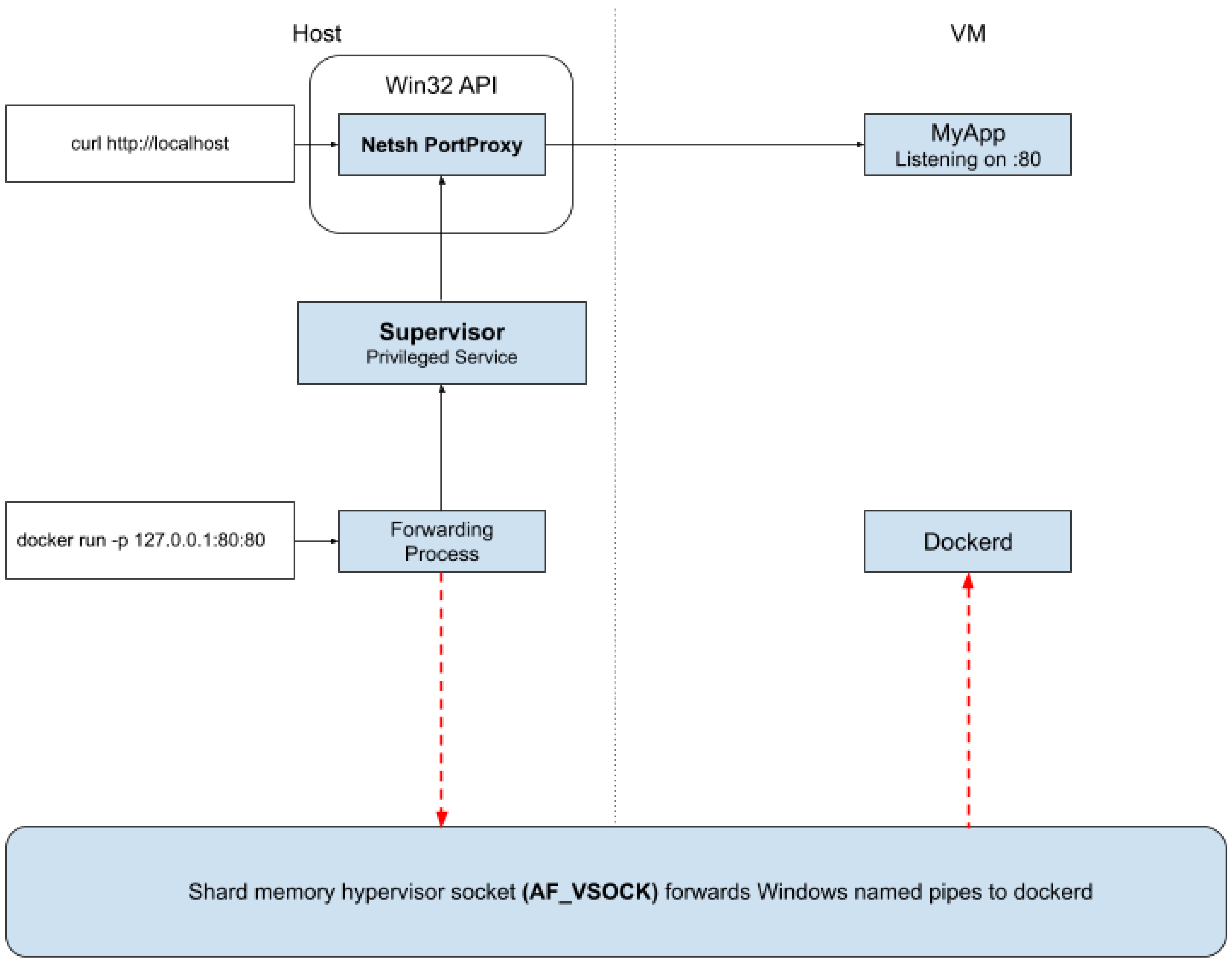
Windows


Host
VM
Public DNS
Corporate VPN
Container A
Container B
Container C
Looking for Container C
Private registry over VPN
Looking for SQL server on the host
coreDNS
host.rancher-desktop.internal:1433
Win32 API
SQL
UDP:53
TCP:53
Host-resolver Host
Host-resolver Peer
Dockerd
TCP and UDP are piped over shared memory hypervisor socket (AF_VSOCK)

Published Ports (Port Binding/Forwarding)

It can be useful to expose ports on the host for various reasons, such as applications wanting to expose UI, accessing an app through a browser, or perhaps even debugging an application that is running in the container.

Windows

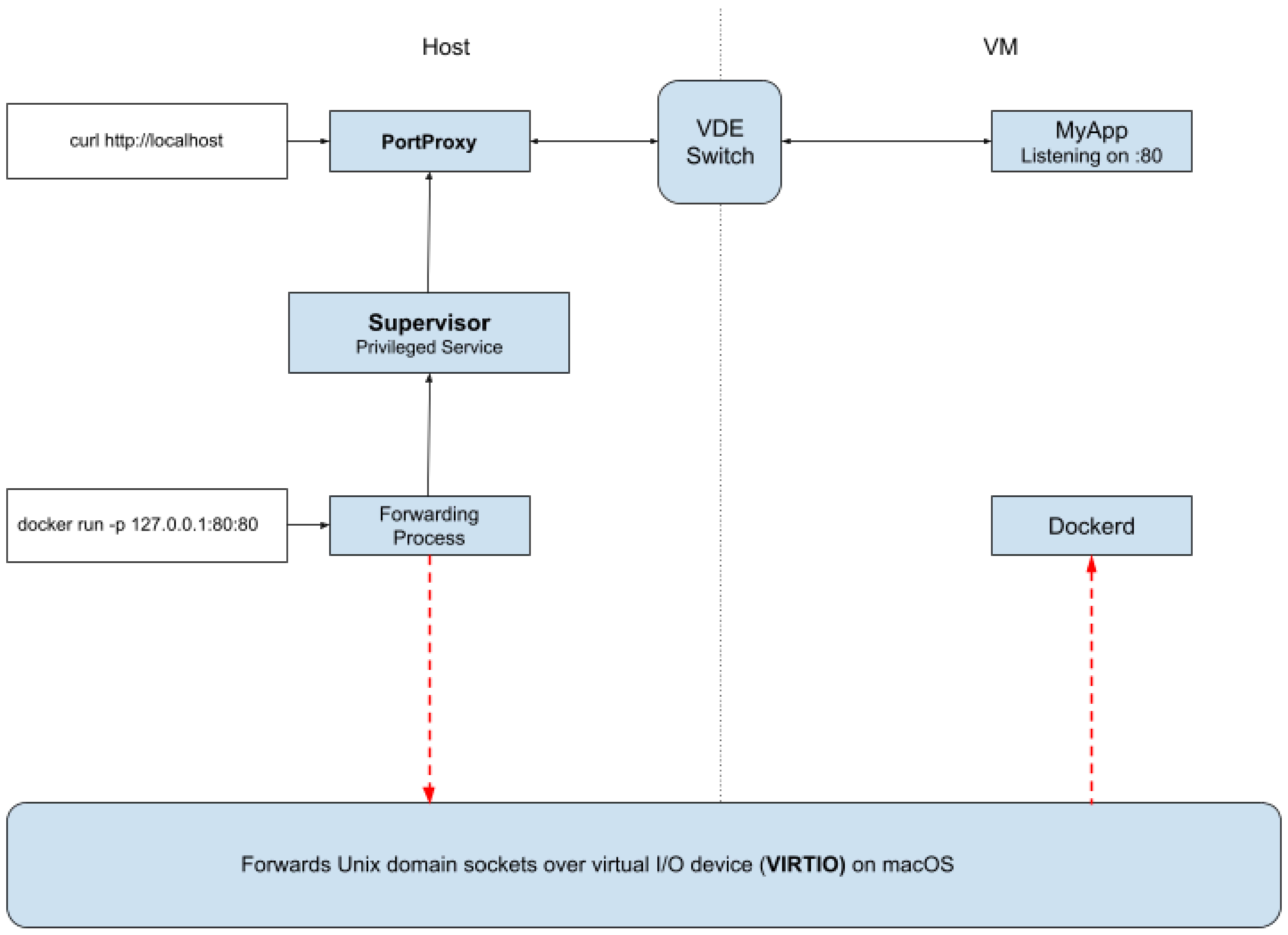


The approach above can be taken to expose the backend container ports on the desired interface if a run command with -p or –publish is issued.

Currently, the forwarding process (as shown in the diagram above) on Windows communicates with dockerd via Windows named pipes over shared memory hypervisor socket address family or more precisely (AF_VSOCK). When a docker API call is issued, the forwarding process (currently wsl-helper) forwards the request to the dockerd. However, in the proposed design above, an additional call will be made to the **Supervisor** process which is a privileged service. When a Supervisor receives the port binding request from the forwarding process, it will then try to publish the desired port on a given IP (if one is not provided, **0.0.0.0** or **INADDR_ANY** is used). If this attempt fails (e.g. if the port is already occupied or if any other failures), a friendly error is displayed to the user.

On Windows, the Supervisor process leverages a built-in utility (netsh interface portproxy) that can be used to provide a port proxy across multiple networks. This API can be used to add or delete the desired port mappings.

macOS


Host
VM
curl http://localhost
PortProxy
VDE Switch
MyApp Listening on :80
Supervisor Privileged Service
docker run -p 127.0.0.1:80:80
Forwarding Process
Dockerd
Forwards Unix domain sockets over virtual I/O device (VIRTIO) on macOS

On macOS a slightly different approach is taken. The **PortProxy** process, as shown in the diagram above, is leveraged in the osx environment to act as a peer stack in the host that represents the process in the lima VM. The PortProxy listens for requests from the Supervisor process which is a privileged helper service that runs as a root from launchd.

Our current architecture and the way we talk to the dockerd on macOS is different from Windows. As mentioned previously above, in Windows the forwarding process for docker API sends the request over secure low-level transport such as shared memory, however, on macOS this process is replaced with ssh tunnel using the bind-address argument(-L). The docker API calls are simply forwarded to dockerd via `ssh -L`. In this proposal, the communication with dockerd will be replaced by a shared memory over Virtual I/O device. All the docker API requests are forwarded as UNIX domain socket connections over a low-level virtual socket address family as demonstrated in the diagram above.

Although on modern macOS, binding to privileged ports on all IPs e.g. **0.0.0.0** or **INADDR_ANY** are an unprivileged operation. However, there is still one case that requires root privilege, which is when the user requests port binding for 127.0.0.1. E.g. `docker run -p 127.0.0.1:80:80.`

Once the PortProxy receives a port mapping request from the Supervisor process, it attempts to map the desired port and IP (if one is provided). If the operation fails (e.g. port is already taken), a friendly error message is displayed to the user. However, if the operation succeeds, the PortProxy starts accepting requests on the specified port and it forwards the requests over a `vde_switch` to the backend process via tap interfaces.

Proxy

As demonstrated in the diagram below, an instance of envoy proxy is provided in the user's host machine. The purpose of this instance is to prevent dockerd restarts and glitches during any changes that occur to the proxy settings or the upstream proxy.

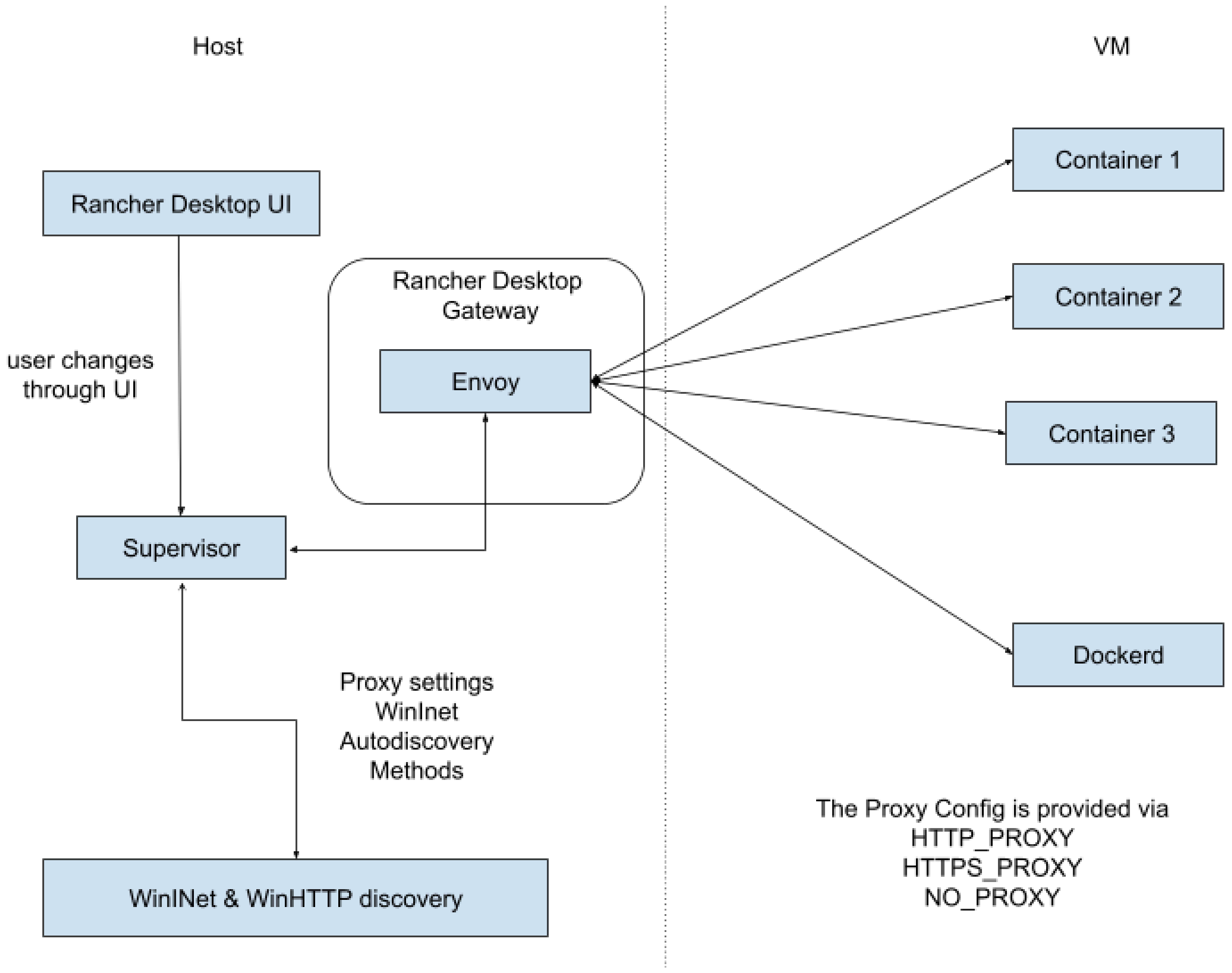
Envoy proxy is a CNCF project and extremely performant that can be configured through a robust API without requiring a restart of the process. This is important as users may change their proxy settings or perhaps eliminate their settings altogether.

The envoy instance can guarantee a complete uptime during these changes. This workflow requires simply pointing the Dockerd to a static IP address on the host via standard proxy environment variables. Any changes to the upstream proxy will then take place dynamically in the envoy instance and the docker engine and containers will continue to operate without any disruptions to the network and users.

It is important to note that the proxy settings which are provided by the user through Rancher Desktop UI will only be consumed by the docker engine and not the containers themselves. This behavior aligns with Docker Desktop. If a user wishes to provide the proxy settings to the individual containers, they can simply provide the proxy settings using the –env argument e.g.

```
docker run \
   --env http_proxy="http://my.proxy.com:3128" \
   --env https_proxy="http://my.proxy.com:3128" \
   nginx sh -c "curl google.com"
```

Windows



On Windows, a user can simply provide the proxy settings using Rancher Desktop UI through the following variables:

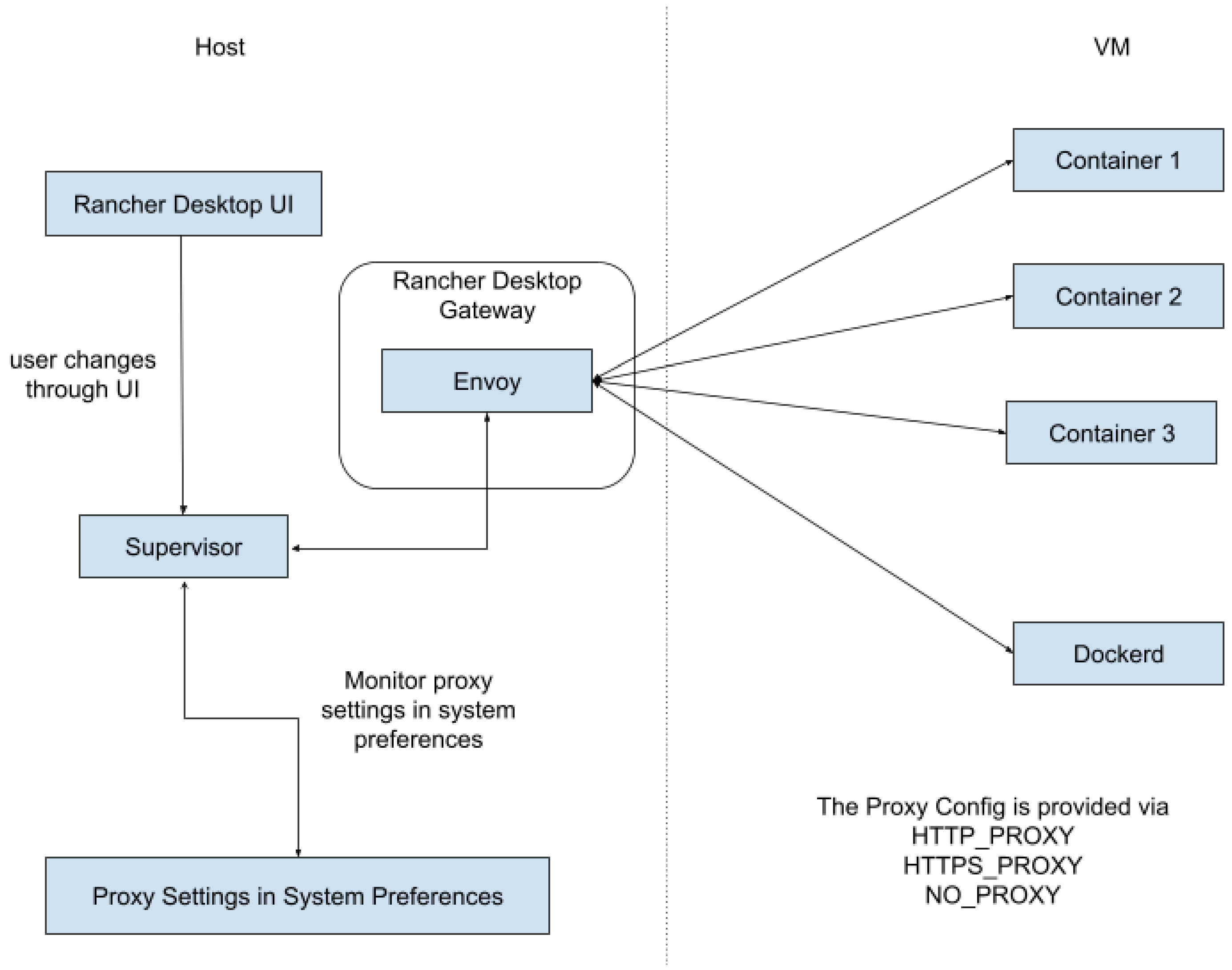
```
HTTP_PROXY - Server endpoint that provides plain http
HTTPS_PROXY - Server endpoint that provides https
FTP_PROXY - Server endpoint that handles ftp
NO_PROXY - List of hosts that can be reached without going through the proxy e.g. localhost, 127.0.0.1
```

Furthermore, if a user wishes to provide authentication for their proxy settings they can simply use the following format, or perhaps we can incorporate a username and password field in the UI.

```
http://<username>:<password>@my.proxy.com:3128/
```

In addition to the manual configuration that a user can provide through the UI, the Supervisor process can also request autodiscovery proxy settings through `WinINet` and `WinHTTP` discovery process using the win32 API. Using the auto-discovery API can provide a more robust workflow for users and easier integration with their Windows corporate authenticated proxy settings. However, this requires additional research and can be tackled as a lower priority item in the roadmap.

macOS


Host
VM
Rancher Desktop UI
user changes through UI
Rancher Desktop Gateway
Envoy
Supervisor
Monitor proxy settings in system preferences
Proxy Settings in System Preferences
Container 1
Container 2
Container 3
Dockerd
The Proxy Config is provided via HTTP_PROXY HTTPS_PROXY NO_PROXY

On macOS, the workflow through the Rancher Desktop will be similar to Windows. However, the auto-discovery of the proxy settings is different. In the macOS, Supervisor process will be watching the system preferences for any changes to the proxy settings associated with the network interfaces. If a change is detected, the Supervisor process will then propagate those changes dynamically in the envoy instance without any action from the user. Otherwise, every other setting and workflow is similar to Windows.