Eden with Software-defined networking (SDN)

Motivation

LF-Edge/Eden is used by EVE developers and EVE community primarily to easily setup and test or experiment with EVE-OS locally. With eden, EVE is running inside a VM and thus does not require any specific hardware device. A released or a custom-built EVE can be very quickly started as a Qemu, Vbox or Parallels VM, configured to connect to an instance of the Adam controller, which is started locally as a docker container. Eden allows testing EVE manually, providing CLIs to build a desired device configuration, dump logs, read and parse published info messages, access EVE OS via ssh/telnet, and more. Additionally, it ships with a test suite used as part of EVE development for regression testing. A subset of this test suite, with the most important and not overly long tests, is being run inside github actions configured for the EVE repository. Every PR must successfully pass eden tests to be allowed for merge.

Apart from locally running EVE VM + Adam, Eden also allows to easily deploy EVE on a Raspberry Pi, or to use Zedcloud instead of Adam. However, the primary use-case is to quickly setup and test EVE in a local environment on commodity hardware.

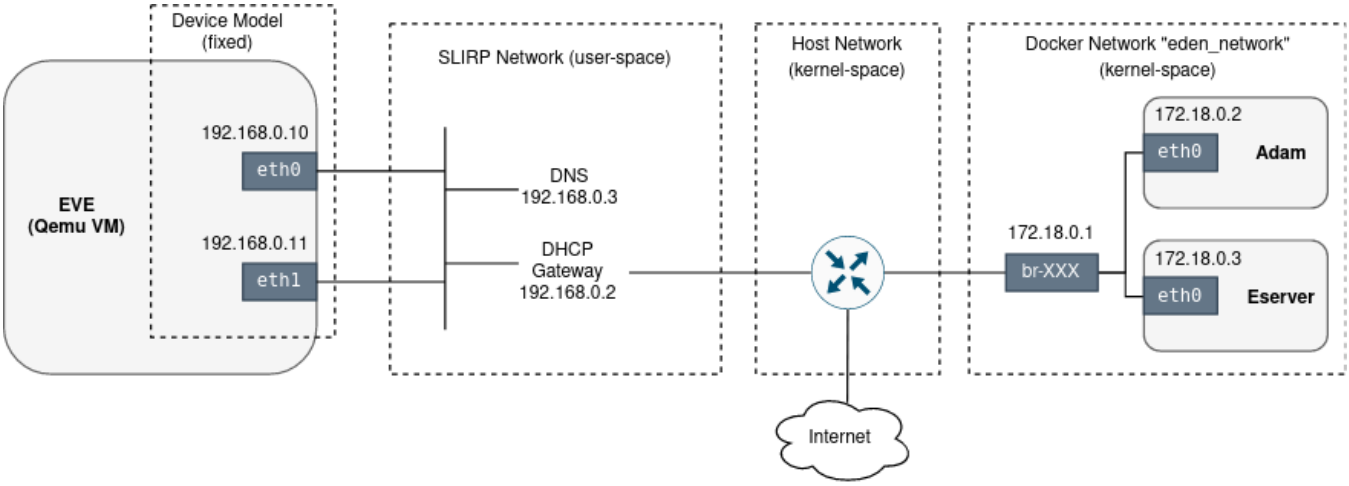
As any software in development, Eden in its present state also has some shortcomings. Some of them, mostly related to ease-of-use, will be covered by a proposed refactoring. The focus of this document is to look at the networking that Eden configures between EVE VM, Adam and other components /endpoints, and propose how this could be enhanced to support EVE testing for a significantly wider range of use-cases wrt. device connectivity. This would include testing of the network config override feature (*DevicePortConfig/override.json*), connectivity with a static IP configuration, proxies, jumbo frames, VLANs, LAGs, VPN network instances, IPv6, multicast, injected network failures, iPXE booting, multiple EVE instances, and more. None of these is currently possible to cover with Eden.

Current Limitations

Eden supports 3 virtualization technologies for running EVE as a VM: Qemu/KVM, VirtualBox and Parallels. In terms of networking configured between EVE and other components of the eden infrastructure, there are small differences between these 3 variants. But in all cases the network configuration is mostly hard-coded: EVE has 1 or 2 ethernet interfaces, DHCP is enabled for both interfaces and DHCP + DNS servers are provided by the virtualization technology.

The Qemu/KVM variant is the most widely used and provides most configuration options. For example, apart from two hard-coded ethernet interfaces (eth0 and eth1) with enabled DHCP, it allows to create an additional tap interface and let the user to manually configure bridging, routing and other aspects of networking for this interface *inside the host network*. Interestingly, eth0 and eth1 are put into the same segment. This is simply because there is one test (*port_forward*) which needs this kind of topology to exercise a specific use-case. In other words, one test from the entire test suite dictates what the hard-coded network topology should be like.

Diagram below shows networking deployed by eden for EVE when QEMU/KVM is used (IP addressing may differ):



The current approach is significantly limiting the scope of networking-related EVE features that can be covered with eden. We are not able to properly test the network config override (override.json), static IP addressing, Proxies, different MTU settings, VLANs, LAGs, network failures, etc. Even DNS and DHCP services as provided by the virtualization technology allow only a very limited scope of customization. In the case of QEMU, we are even using a user-space SLIRP network, which additionally limits the observability and what we can do with the network (e.g. not possible to ping the device).

Moreover, traffic between EVE and Adam/Eserver is passing through and depends on the host network, which is not ideal. It is preferred to have the testing infrastructure, including networking, more isolated in order to avoid interference with the host.

Finally, to do at least some more advanced network testing with the current setup, we run network services (like DHCP server) in some eden tests as EVE applications. The problem is that they require NET_ADMIN capabilities. For security reasons, EVE does not grant these capabilities to native containers, therefore it is required to deploy apps as VMs-in-Containers for these tests to pass. Without HW-assisted virtualization, these tests must be therefore skipped. This is the case in github workflows. In other words, some eden tests are not being run automatically by github for regression.

Software-Defined Networking

The proposal of this document is to make the networking deployed by eden *fully programmable* and also more *isolated* from the host, allowing to exercise a much wider range of network-oriented tests and experiments (without tampering with the host network).

Following the principles of the Software-Defined Networking (SDN for short), a desired state of the network topology will be described *declaratively* and applied into the network stack *programmatically* using a *management agent*. In our case, the state will be modeled using Go structures (we can also consider protobuf) and submitted into the agent in JSON format via RESTful APIs. The agent will be responsible to reconcile the state of the network stack with the last-submitted desired state.

In EVE OS, the hardware-side of the connectivity is covered by the so-called “Device Model”. Similarly, we could call the networking around EVE VM as “Network model” (just for eden). The network model will affect the device model - for example it will determine the number of ethernet interfaces. It means that changes to a network model may require restarting EVE VM with a changed device model (this is already possible with eden). Changes not affecting the emulated EVE VM NICs, the SDN should be able to apply in the runtime without restarting EVE.

Eden-SDN

We propose to develop a new component for eden infrastructure, running as an additional light-weight VM, built using linuxkit. It will run a management agent, written in Go. The main challenge for this agent is the reconciliation between the desired state and the actual state of the network stack. There is already a tool for this task inside the EVE repository: State Reconciler.

Eden-SDN will use Linux network stack and the tools that it provides to build a desired network topology (network namespaces, VETHs, Linux bridge, VLAN sub-interfaces, iptables, etc.). Additionally, it will use dnsmasq as a DNS and DHCP server, and goproxy as a MITM or an explicit HTTP/HTTP /SOCKS proxy. If VPN network instances ever become a priority, we can include strongSwan as an IPsec gateway. However, due to a lack of emulation support, the SDN will not cover wireless connectivity (WiFi, cellular).

By encapsulating all these aspects of networking into a VM, we will get better isolation from the host. However, adam controller will still run as a container on the host inside the docker network. And, of course, traffic headed toward the Internet will also cross the host network.

The agent will run an HTTP server and expose RESTful endpoints to apply/get network model, get status and more. These endpoints will be used by eden CLIs and after the refactor also by the eden library.

Integration

Firstly, the eden-sdn VM will be built inside the eden repository using linuxkit CLI during the “eden setup” stage. It will be deployed by “eden start” using the same method as EVE VM.

In terms of supported virtualization technologies, the focus will be on the Qemu/KVM variant. Others (Vbox, Parallels) could be unsupported by the SDN for now and we just need to ensure that the existing behavior will remain unaffected.

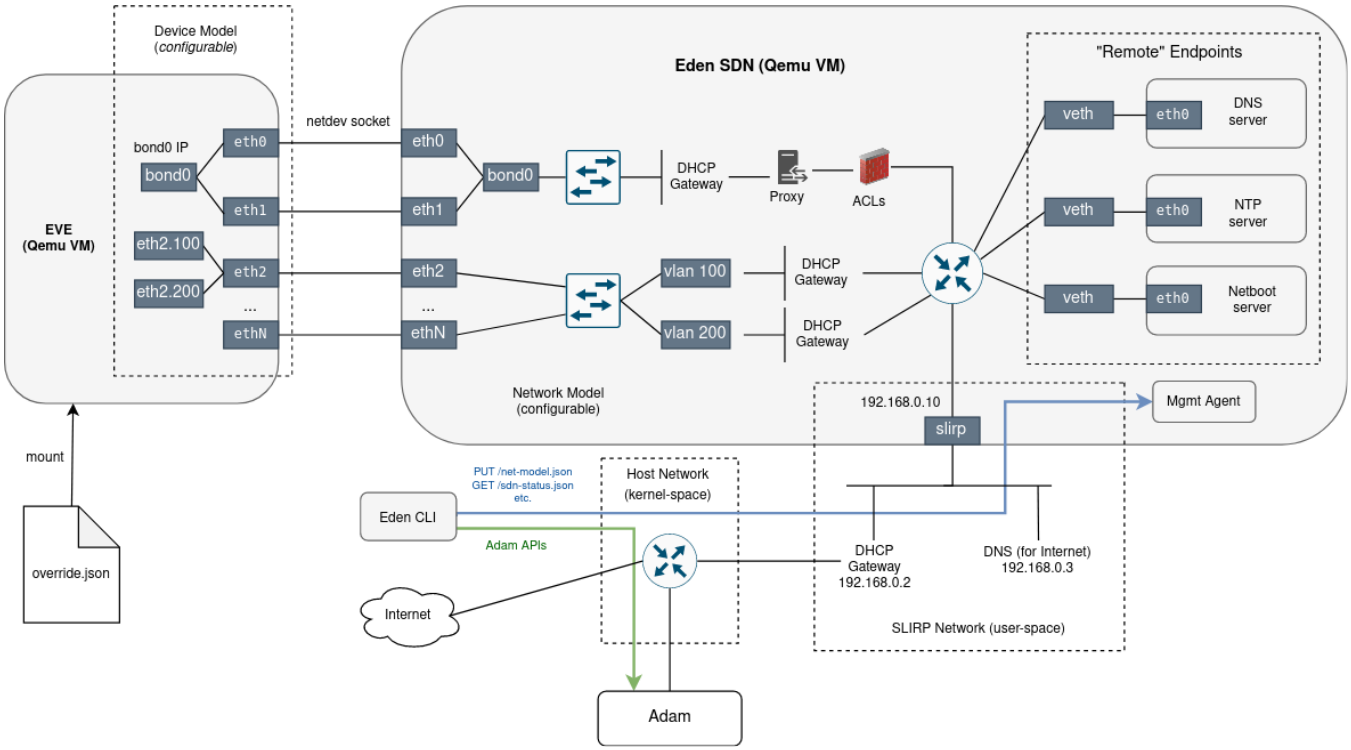
With Qemu, the socket networking backend will be used to create ethernet connections between EVE and SDN VMs.

The network model will be configurable in the same way as the device model - using a configuration option pointing to a JSON file:

```
eden config set $EDEN_CONFIG –key sdn.network-model
eden config set $EDEN_CONFIG –key eve.devmodelfile
```

Note that once eden is refactored, eden tests will be able to operate with the Go structures of the model directly, instead of writing/reading the JSON configuration into/from a file.

Diagram below shows an integration of the eden-sdn VM with the rest of the eden infrastructure. Note that the network topology displayed is only an example. In practice it can be anything that can be modeled by the network model (see “Network Model” section below).



Zedcloud

In theory, Eden-SDN could be also used in between EVE and Zedcloud to add extra network components, like proxies, LAGs, VLANs, etc., into the path (eden allows to deploy EVE VM for onboarding with Zedcloud instead of Adam). A request sent from EVE towards zedcloud will first traverse Eden-SDN, exercising whatever was programmed by the Network model, followed by a (S-NATed) hop into the host network and continuing with the original path. Response will traverse the same path but in the reverse order.

RESTful APIs

Methods provided by the management agent of the eden-sdn. Note that they will be accessed by eden CLIs (and the library) via Qemu-forwarded port.

GET localhost:$EDEN_SDN_PORT**/net-model.json** • Response Content-Type: application/json
GET localhost:$EDEN_SDN_PORT**/sdn-status.json** • Response Content-Type: application/json
GET localhost:$EDEN_SDN_PORT**/net-config.gv** • Response Content-Type: text/vnd.graphviz
etc. (metrics)

Network Model

Model describing the desired network topology and the configuration declaratively. We propose to use Go structures to build the model. Protobuf is also a good alternative, but likely not necessary for this case.

```
// NetworkModel is used to declaratively describe the intended state of the networking
// around EVE VM(s) for the testing purposes. The model is submitted in the JSON format
// to Eden-SDN Agent, running inside a separate VM, connected to EVE VM(s) via inter-VM
// network interfaces, and emulating the desired networking using the Linux network stack,
// network namespaces, netfilter and with the help of several open-source projects,
// such as dnsmasq, radvd, goproxy, etc.
type NetworkModel struct {
```

```
    // Ports : network interfaces connecting EVE VM(s) with Eden-SDN VM.
    // Each port is essentially an interconnected pair of network interfaces, with one side
    // inserted into the EVE VM and the other to Eden-SDN. These interface pairs are created
    // in both VMs in the order as listed here. This means that Ports[0] will appear
    // as the first interface in EVE (likely named "eth0" by the kernel) and likewise
    // as the first interface in Eden-SDN. Note that Eden-SDN will have one more extra
    // interface, added as last and used for management (for eden to talk to SDN mgmt agent).
    Ports []Port `json:"ports"`
    // Bonds are aggregating multiple ports for load-sharing and redundancy purposes.
    Bonds []Bond `json:"bonds"`
    // Bridges provide L2 connectivity.
    Bridges []Bridge `json:"bridges"`
    // Networks provide L3 connectivity.
    Networks []Network `json:"networks"`
    // Endpoints simulate "remote" clients and servers.
    Endpoints Endpoints `json:"endpoints"`
    // Firewall is applied between Networks, Endpoints and the outside of Eden-SDN
    // (controller, Internet).
    Firewall Firewall `json:"firewall"`
    // Host configuration that Eden-SDN is informed about.
    // Eden SDN needs to learn about the host configuration to properly route traffic
    // to the controller and beyond to the Internet.
    // If not defined (nil pointer), Eden will try to detect host config automatically.
    Host *HostConfig `json:"host,omitempty"`
}

// Port is a network interface connecting EVE VM with Eden-SDN VM.
type Port struct {
    // LogicalLabel : logical name used for reference.
    LogicalLabel string `json:"logicalLabel"`
    // MAC address assigned to the interface on the SDN side.
    // If not specified by the user, Eden will generate a random MAC address.
    MAC string `json:"mac"`
    // MTU : Maximum transmission unit (for the SDN side).
    MTU uint16 `json:"mtu"`
    // AdminUP : whether the interface should be UP on the SDN side.
    // Put down to test link-down scenarios on EVE.
    AdminUP bool `json:"adminUP"`
    // EVEConnect : plug the other side of the port into a given EVE instance.
    EVEConnect EVEConnect `json:"eveConnect"`
}

// EVEConnect : connects Port to a given EVE instance.
type EVEConnect struct {
    // EVEInstance : name of the EVE instance to which a given port is connected.
    // In the future, Eden may support running multiple EVE instances connected
    // to the same SDN and controller. It is likely that each such instance
    // will be assigned a unique logical label, which this field will reference.
    // However, currently Eden is only able to manage a single EVE instance.
    // For the time being it is therefore expected that this field is empty
    // and EVEConnect refers to the one and only EVE instance.
    EVEInstance string `json:"eveInstance"`
    // MAC address assigned to the interface on the EVE side.
    // If not specified by the user, Eden will generate a random MAC address.
    MAC string `json:"mac"`
}

// Bond is aggregating multiple ports for load-sharing and redundancy purposes.
// Also known as Link aggregation group (LAG).
type Bond struct {
        // LogicalLabel : logical name used for reference.
        LogicalLabel string `json:"logicalLabel"`
        // Ports : list of aggregated ports, referenced by logical labels.
        Ports []string `json:"ports"`
        // Mode : bonding policy.
        Mode BondMode `json:"mode"`
        // LacpRate : LACPDU packets transmission rate.
        // Applicable for BondMode802Dot3AD only.
        LacpRate LacpRate `json:"lacpRate"`
        // MIIMonitor : MII link state monitoring.
        // Link monitoring is either disabled or one of the monitors
```

```
        // is enabled (MII or ARP), never both at the same time.
        MIIMonitor BondMIIMonitor `json:"miiMonitor"`
        // ARPMonitor : ARP-based link state monitoring.
        // Link monitoring is either disabled or one of the monitors
        // is enabled (MII or ARP), never both at the same time.
        ARPMonitor BondArpMonitor `json:"arpMonitor"`
}

// BondMIIMonitor : MII link monitoring parameters.
type BondMIIMonitor struct {
        // Enabled : set to true to enable MII.
        Enabled bool `json:"enabled"`
        // Interval specifies the MII link monitoring frequency in milliseconds.
        // This determines how often the link state of each bond slave is inspected
        // for link failures.
        Interval uint32 `json:"interval"`
        // UpDelay specifies the time, in milliseconds, to wait before enabling
        // a bond slave after a link recovery has been detected.
        // The UpDelay value should be a multiple of the monitoring interval; if not,
        // it will be rounded down to the nearest multiple.
        // The default value is 0.
        UpDelay uint32 `json:"upDelay"`
        // DownDelay specifies the time, in milliseconds, to wait before disabling a bond
        // slave after a link failure has been detected.
        // The DownDelay value should be a multiple of the monitoring interval; if not,
        // it will be rounded down to the nearest multiple.
        // The default value is 0.
        DownDelay uint32 `json:"downDelay"`
}

// BondArpMonitor : ARP-based link monitoring parameters.
type BondArpMonitor struct {
        // Enabled : set to true to enable ARP-based link monitoring.
        Enabled bool `json:"enabled"`
        // Interval specifies the ARP link monitoring frequency in milliseconds.
        Interval uint32 `json:"interval"`
        // IPTargets specifies the IPv4 addresses to use as ARP monitoring peers.
        // These are the targets of ARP requests sent to determine the health of links.
        IPTargets []string `json:"ipTargets"`
}

// LacpRate specifies the rate in which bond driver will ask LACP link partners
// to transmit LACPDU packets in 802.3ad mode.
type LacpRate uint8

const (
        // LacpRateSlow : Request partner to transmit LACPDUs every 30 seconds.
        // This is the default rate.
        LacpRateSlow LacpRate = iota
        // LacpRateFast : Request partner to transmit LACPDUs every 1 second.
        LacpRateFast
)

// BondMode specifies the policy indicating how bonding slaves are used
// during network transmissions.
type BondMode uint8

const (
        // BondModeBalanceRR : Round-Robin
        // This is the default mode.
        BondModeBalanceRR BondMode = iota
        // BondModeActiveBackup : Active/Backup
        BondModeActiveBackup
        // BondModeBalanceXOR : select slave for a packet using a hash function
        BondModeBalanceXOR
        // BondModeBroadcast : send every packet on all slaves
        BondModeBroadcast
        // BondMode802Dot3AD : IEEE 802.3ad Dynamic link aggregation
        BondMode802Dot3AD
        // BondModeBalanceTLB : Adaptive transmit load balancing
        BondModeBalanceTLB
```

```
        // BondModeBalanceALB : Adaptive load balancing
        BondModeBalanceALB
)

// Bridge provides L2 connectivity.
type Bridge struct {
    // LogicalLabel : logical name used for reference.
    LogicalLabel string `json:"logicalLabel"`
    // Logical labels of ports.
    Ports []string `json:"ports"`
    // Logical labels of bonds.
    Bonds []string `json:"bonds"`
}

// Network provides L3 connectivity.
type Network struct {
    // LogicalLabel : logical name used for reference.
    LogicalLabel string `json:"logicalLabel"`
    // Logical label of a Bridge to which the network is attached.
    Bridge string `json:"bridge"`
    // Leave zero value to express intent of not using VLAN for this network.
    VlanID uint16 `json:"vlanID"`
    // Subnet : network address + netmask (IPv4 or IPv6).
    Subnet string `json:"subnet"`
    // GwIP should be inside the Subnet.
    GwIP string `json:"gwIP"`
    // DHCP configuration.
    DHCP DHCP `json:"dhcp"`
    // TransparentProxy is a proxy that both HTTP and HTTPS traffic is forwarded through
    // transparently, before it reaches router, endpoints, firewall, etc.
    TransparentProxy *Proxy `json:"transparentProxy,omitempty"`
    // Router configuration. Every network has a separate routing context.
    // Undefined (nil) means that everything should be routed and accessible.
    // That includes all networks, endpoints and the outside of Eden SDN.
    Router *Router `json:"router,omitempty"`
}

// DHCP configuration.
// Note that for IPv6 we prefer to use Router Advertisement over DHCPv6 to publish
// all this information to hosts on the network.
// But DHCPv6 is still needed and used to convey NTP and netboot configuration (if provided).
type DHCP struct {
    // Enable DHCP. Set to false to use EVE with static IP addressing.
    Enable bool `json:"enable"`
    // IPRange : a range of IP addresses to allocate from.
    // Not applicable for IPv6.
    IPRange IPRange `json:"ipRange"`
    // DomainName : name of the domain assigned to the network.
    // It is propagated to clients using the DHCP option 15 (24 in DHCPv6).
    DomainName string `json:"domainName"`
    // DNSClientConfig : DNS configuration passed to clients via DHCP.
    DNSClientConfig
    // Public NTP server to announce via DHCP option 42 (56 in DHCPv6).
    // Do not configure both PublicNTP and PrivateNTP.
    PublicNTP string `json:"publicNTP"`
    // Logical label of an NTP endpoint running inside Eden SDN, announced to client
    // via DHCP option 42 (56 in DHCPv6).
    // Do not configure both PublicNTP and PrivateNTP.
    PrivateNTP string `json:"privateNTP"`
    // WPAD : URL with a location of a PAC file, announced using the Web Proxy Auto-Discovery
    // Protocol (WPAD) and DHCP.
    // The PAC file should contain a javascript that the client will use to determine
    // which proxy to use for a given request.
    // URL example: http://wpad.example.com/wpad.dat
    // The client will learn the PAC file location using the DHCP option 252.
    // An alternative approach is to use DNS (with a DNSServer endpoint).
    WPAD string `json:"wpad"`
    // NetbootServer : Logical label of a NetbootServer endpoint which the client should use
    // to boot EVE OS from. The IP address or FQDN and the provisioning file (iPXE script)
    // location will be announced to the client using DHCP options 66 and 67 (59 in DHCPv6).
    // Eden-SDN will announce either IP address or FQDN depending on whether any of the
```

```
    // assigned private DNS servers is able to resolve the NetbootServer domain name.
    NetbootServer string `json:"netbootServer"`
}

// DNSClientConfig : DNS configuration for a client.
type DNSClientConfig struct {
    // PublicDNS : list of IP addresses of public DNS servers to announce via DHCP option 6.
    // For example: ["1.1.1.1", "8.8.8.8"]
    PublicDNS []string `json:"publicDNS"`
    // PrivateDNS : list of DNS servers running as endpoints inside Eden SDN,
    // announced to clients via DHCP option 6.
    // The list should contain logical labels of those endpoints, not IP addresses!
    PrivateDNS []string `json:"privateDNS"`
}

// Proxy can be either transparent or configured explicitly.
type Proxy struct {
    // CertPEM : Proxy certificate of the certificate authority in the PEM format.
    // Proxy will use CA cert to sign certificate that it generates for itself.
    // EVE should be configured to trust CA certificate.
    // Not needed if proxy is just forwarding all flows (i.e. not terminating TLS).
    CACertPEM string `json:"caCertPEM"`
    // CAKeyPEM : Proxy key of the certificate authority in the PEM format.
    // Proxy will use CA cert to sign certificate that it generates for itself.
    // EVE should be configured to trust CA certificate.
    // Not needed if proxy is just forwarding all flows (i.e. not terminating TLS).
    CAKeyPEM string `json:"caKeyPEM"`
    // ProxyRules : a set of rules that decides what to do with proxied traffic.
    // By default (no rules defined), proxy will just forward all the flows.
    ProxyRules []ProxyRule `json:"proxyRules"`
}

// ProxyRule : rule used by a proxy, which, if matches a given flow, decides what
// to do with it.
type ProxyRule struct {
    // ReqHost : host from HTTP request header (or from the SNI value in the TLS ClientHello)
    // to match this rule with (e.g. "google.com").
    // Empty ReqHost should be used with the default rule (put one at most).
    ReqHost string `json:"reqHost"`
    // Action to take.
    Action ProxyAction `json:"action"`
}

// Router routing traffic for a network based on the reachability requirements.
type Router struct {
    // OutsideReachability : If enabled then it is possible to use the network to access
    // endpoints outside of Eden SDN (unless blocked by firewall).
    // This includes the controller (Adam, zedcloud), eserver (image cache) and the Internet.
    OutsideReachability bool `json:"outsideReachability"`
    // ReachableEndpoints : Logical labels of reachable endpoints.
    ReachableEndpoints []string `json:"reachableEndpoints"`
    // ReachableNetworks : Logical labels of reachable networks.
    ReachableNetworks []string `json:"reachableNetworks"`
}

// Note that traffic not matched by any rule is allowed!
type Firewall struct {
    // Rules : firewall rules applied in the order as configured.
    Rules []FwRule `json:"rules"`
}

// FwRule : a firewall rule.
type FwRule struct {
    // SrcSubnet : subnet to match the source IP address with.
    SrcSubnet string `json:"srcSubnet"`
    // DstSubnet : subnet to match the destination IP address with.
    DstSubnet string `json:"dstSubnet"`
    // Protocol : filter by protocol.
    Protocol FwProto `json:"protocol"`
    // Ports : list of destination port to which the rule applies.
    // Empty = any.
```

```
    Ports []uint16 `json:"ports"`
    // Action to take.
    Action FwAction `json:"action"`
}

// HostConfig : host configuration that Eden-SDN needs to be informed about.
type HostConfig struct {
    // HostIPs : list of IP addresses used by the host system (on top of which
    // Eden runs).
    // Eden SDN requires at least one routable host IP address.
    HostIPs []string `json:"hostIPs"`
    // NetworkType : which IP versions are used by the host.
    // Even if host uses IPv4 only, it is still possible to have IPv6 inside
    // Eden-SDN. Connectivity between EVE (IPv6) and the controller (IPv4) is established
    // automatically using DNS64 and NAT64. However, the opposite case (from IPv4 to IPv6)
    // is not supported. In other words, to test EVE with IPv4, it is required for the host
    // to use IPv4 (single or dual stack).
    NetworkType NetworkType `json:"networkType"`
}

// Endpoints simulate "remote" clients and servers.
type Endpoints struct {
        // Clients : list of clients. Can be used to run requests towards EVE.
        Clients []Client `json:"clients,omitempty"`
        // DNSServers : list of DNS servers. Can be referenced in DHCP.PrivateDNS.
        DNSServers []DNSServer `json:"dnsServers,omitempty"`
        // NTPServers : list of NTP servers. Can be referenced in DHCP.PrivateNTP.
        NTPServers []NTPServer `json:"ntpServers,omitempty"`
        // HTTPServers : list of HTTP(s) servers. Can be used to test HTTP(s) connectivity
        // from EVE, to serve PAC files, etc.
        HTTPServers []HTTPServer `json:"httpServers,omitempty"`
        // ExplicitProxies : proxies that must be configured explicitly.
        // Consider using together with NetworkModel.Firewall, configured to block HTTP(s)
        // traffic that tries to bypass a proxy.
        ExplicitProxies []ExplicitProxy `json:"explicitProxies,omitempty"`
        // NetbootServers : HTTP/TFTP servers providing artifacts needed to boot EVE OS
        // over a network (using netboot/PXE + iPXE).
        NetbootServers []NetbootServer `json:"netbootServers,omitempty"`
}

// Endpoint simulates "remote" client or a server.
type Endpoint struct {
        // LogicalLabel : logical name used for reference.
        LogicalLabel string `json:"logicalLabel"`
        // FQDN : Fully qualified domain name of the endpoint.
        FQDN string `json:"fqdn"`
        // Subnet : network address + netmask (IPv4 or IPv6).
        // Subnet needs to fit at least two host IP addresses,
        // one for the endpoint, another for a gateway.
        Subnet string `json:"subnet"`
        // IP should be inside of the Subnet.
        IP string `json:"ip"`
        // MTU of the endpoint's interface.
        MTU uint16 `json:"mtu"`
}

// Client emulates a remote client.
// Can be used to run requests towards EVE.
type Client struct {
        // Endpoint configuration.
        Endpoint
}

// DNSServer : endpoint providing DNS service.
type DNSServer struct {
        // Endpoint configuration.
        Endpoint
        // StaticEntries : list of FQDN->IP entries statically configured
        // for the server. These are typically used for endpoints running inside Eden-SDN,
        // which are obviously not known to the public DNS servers.
        StaticEntries []DNSEntry `json:"staticEntries"`
```

```
        // UpstreamServers : list of IP addresses of public DNS servers to forward
        // requests to (unless there is a static entry).
        UpstreamServers []string `json:"upstreamServers"`
}

// DNSEntry : Mapping between FQDN and an IP address.
type DNSEntry struct {
        // FQDN : Fully qualified domain name.
        // Can be a reference to endpoint FQDN:
        //  - "endpoint-fqdn.<endpoint-logical-label>" - translated to endpoint's FQDN by Eden-SDN
        FQDN string `json:"fqdn"`
        // IP address or a special value that Eden-SDN will automatically translate
        // to the corresponding IP address:
        //  - "endpoint-ip.<endpoint-logical-label>" - translated to IP address of the endpoint
        //  - "adam-ip" - translated to IP address on which Adam (open-source controller) is deployed and
accessible
        IP string `json:"ip"`
}

// HTTPServer : HTTP(s) server.
type HTTPServer struct {
        // Endpoint configuration.
        Endpoint
        // DNSClientConfig : DNS configuration to be applied for the HTTP server.
        DNSClientConfig
        // HTTPPort : port to listen for HTTP requests.
        // Zero value can be used to disable HTTP.
        HTTPPort uint16 `json:"httpPort"`
        // HTTPSPort : port to listen for HTTPS requests.
        // Zero value can be used to disable HTTPS.
        HTTPSPort uint16 `json:"httpsPort"`
        // CertPEM : Server certificate in the PEM format. Required for HTTPS.
        CertPEM string `json:"certPEM"`
        // KeyPEM : Server key in the PEM format. Required for HTTPS.
        KeyPEM string `json:"keyPEM"`
        // Maps URL Path to a content to be returned inside the HTTP(s) response body
        // (text/plain content type).
        Paths map[string]HTTPContent `json:"paths"`
}

// HTTPContent : content returned by an HTTP(s) handler.
type HTTPContent struct {
        // ContentType : HTTP(S) Content-Type.
        ContentType string `json:"contentType"`
        // Content : content returned inside a HTTP(s) response body.
        // It is a string, so binary content is not possible for now.
        Content string `json:"content"`
}

// NTPServer : NTP server.
type NTPServer struct {
        // Endpoint configuration.
        Endpoint
        // List of (public) NTP servers to synchronize with, each referenced
        // by an IP address or a FQDN.
        UpstreamServers []string `json:"upstreamServers"`
}

// ExplicitProxy : HTTP(S) proxy configured explicitly.
type ExplicitProxy struct {
        // Endpoint configuration.
        Endpoint
        // Proxy configuration (common to transparent and explicit proxies).
        Proxy
        // DNSClientConfig : DNS configuration to be applied for the proxy.
        DNSClientConfig
        // HTTPPort : HTTP proxy port.
        // Zero value can be used to disable HTTP proxy.
        HTTPPort uint16 `json:"httpPort"`
        // HTTPSPort : HTTPS proxy port.
        // Zero value can be used to disable HTTPS proxy.
```

```
        HTTPSPort uint16 `json:"httpsPort"`
        // Users : define for username/password authentication, leave empty otherwise.
        Users []UserCredentials `json:"users"`
}

// UserCredentials : User credentials for an explicit proxy.
type UserCredentials struct {
        // Username
        Username string `json:"username"`
        // Password
        Password string `json:"password"`
}

// NetbootServer provides HTTP and TFTP server endpoints, serving all artifacts
// needed to boot EVE OS over a network (using iPXE, potentially also supporting
// older PXE-only clients).
// Use in combination with DHCP (see DHCP.NetbootServer).
// XXX Note that in all likelihood, TFTP will serve an iPXE (UEFI) bootloader,
// that once booted will download and boot EVE artifacts over the HTTP endpoint.
// This can work with only a little magic in the DHCP server configuration,
// known as chainloading [1].
// If a client only understands netboot/PXE, DHCP will point the client first
// to the TFTP endpoint. Once the client has booted iPXE, it will be directed
// by the DHCP server to the iPXE script from the HTTP endpoint (just like any other
// iPXE-enabled client).
//
// Example config for dnsmasq:
//   # Boot for iPXE. The idea is to send two different
//   # filenames, the first loads iPXE, and the second tells iPXE what to
//   # load. The dhcp-match sets the ipxe tag for requests from iPXE.
//   #dhcp-boot=undionly.kpxe
//   #dhcp-match=set:ipxe,175 # iPXE sends a 175 option.
//   #dhcp-boot=tag:ipxe,http://boot.ipxe.org/demo/boot.php
//
// [1] https://ipxe.org/howto/chainloading
type NetbootServer struct {
        // Endpoint configuration.
        Endpoint
        // TFTPArtifacts : boot artifacts served by the TFTP server.
        // If not specified, Eden will automatically put iPXE bootloader
        // as the entrypoint.
        TFTPArtifacts []NetbootArtifact `json:"tftpArtifacts"`
        // HTTPArtifacts : boot artifacts served by the HTTP server.
        // If not specified, Eden will automatically put iPXE artifacts
        // needed to boot EVE OS (as links to eserver where these artifacts
        // are uploaded).
        HTTPArtifacts []NetbootArtifact `json:"httpArtifacts"`
}

// NetbootArtifact - one of the artifacts used to boot EVE OS over a network.
type NetbootArtifact struct {
        // Filename : name of the file.
        // It will be served by the associated NetbootServer at the endpoint:
        // (http|tftp)://<netboot-server-fqdn>/<Filename>
        Filename string `json:"filename"`
        // DownloadFromURL : HTTP URL from where the artifact will be downloaded
        // by the netboot server. It can for example point to the eserver.
        // Note that Netboot server will forward the artifact content, not redirect
        // to this URL.
        DownloadFromURL string `json:"downloadFromURL"`
        // Entrypoint : Is this the entrypoint for netboot (i.e. the artifact to boot from)?
        // In case of iPXE, this would be enabled for the inital iPXE script.
        // Exactly one NetbootArtifact should be marked as entrypoint inside
        // both lists NetbootServer.TFTPArtifacts and NetbootServer.HTTPArtifacts.
        // If enabled, this file is then announced to netboot clients using DHCP
        // option 67 (59 in DHCPv6).
        Entrypoint bool `json:"entrypoint"`
}

// IPRange : a range of IP addresses.
type IPRange struct {
```

```
    // FromIP : start of the range (includes the address itself).
    FromIP string `json:"fromIP"`
    // ToIP : end of the range (includes the address itself).
    ToIP string `json:"toIP"`
}

// NetworkType : type of the network wrt. IP version used.
type NetworkType uint8

const (
    // Ipv4Only : host uses IPv4 only.
    Ipv4Only NetworkType = iota
    // Ipv6Only : host uses IPv6 only.
    Ipv6Only
    // DualStack : host tuns with dual stack.
    DualStack = 8
)

// ProxyAction : proxy action.
type ProxyAction uint8

const (
    // PxForward : just forward proxied traffic.
    PxForward ProxyAction = iota
    // PxReject : reject (block) proxied traffic.
    PxReject
    // PxMITM : act as a man-in-the-middle (split TLS tunnel in two).
    PxMITM
)

// FwAction : firewall action.
type FwAction uint8

const (
    // Allow traffic.
    FwAllow FwAction = iota
    // Deny traffic.
    FwDeny
)

// FwProto : protocol to apply a firewall rule on.
type FwProto uint8

const (
    AnyProto FwProto = iota
    ICMP
    TCP
    UDP
)
```