# Romana Documentation Documentation

*Release Release v2.0 Documentation*

Romana Team

Jun 20, 2018

# Contents

CHAPTER 1

# Welcome to Romana

Romana is a network and security automation solution for cloud native applications.

- Romana automates the creation of isolated cloud native networks and secures applications with a distributed firewall that applies access control policies consistently across all endpoints (pods or VMs) and services, wherever they run.
- Through Romana's topology aware IPAM, endpoints receive natively routable addresses: No overlays or tunnels are required, increasing performance and providing operational simplicity.
- Because IP addresses are assigned with network topology in mind, routes within the network are highly aggregated, reducing the impact on networking hardware, and allowing more secure configurations.
- Supports Kubernetes and OpenStack clusters, on premise or on AWS.

## 1.1 Quick Start

To get started with Romana on Kubernetes, go here.

For OpenStack installations, please contact us by email or on Slack.

CHAPTER 2

# Installation

For clusters created with kops or kubeadm with default settings, predefined YAML files are provided so that you can install easily by using kubectl apply. If you are not using the default settings, some changes to the YAML files will be required - see the *notes*, below.

If you have made your own customized installation of Kubernetes or used a different tool to create the cluster, then you should refer to the detailed components page, and align the example configuration with the details specific to your cluster.

## 2.1 Installation using kubeadm

Follow the Kubernetes cluster configuration guide for Using kubeadm to Create a Cluster, and complete steps 1 and 2. Then, to install Romana, run

```
kubectl apply -f https://raw.githubusercontent.com/romana/romana/master/docs/
↪kubernetes/romana-kubeadm.yml
```

Please see special notes below if - you are using a non-default range for Kubernetes Service IPs - want to specify your own IP range for Pod IPs - are running in virtualbox - have cluster nodes in multiple subnets

## 2.2 Installation with kops

As of kops v1.8, Romana is a built-in CNI networking provider that can be installed directly by folloing the kops documentation.

If you are using an earlier version of kops, Romana can be installed by using the --networking cni option. You will need to SSH directly to your master node to install Romana after the cluster has finished launching.

```
# Connect to the master node
ssh admin@master-ip
```

(continues on next page)

(continued from previous page)

```
# Check that Kubernetes is running and that the master is in NotReady state
kubectl get nodes
```

You should see output similar to the below example.

```
NAME                                          STATUS             AGE         VERSION
ip-172-20-xx-xxx.us-west-2.compute.internal   NotReady,master    2m          v1.7.0
```

Then, to install Romana, run

```
kubectl apply -f https://raw.githubusercontent.com/romana/romana/master/docs/
↪kubernetes/romana-kops.yml
```

It will take a few minutes for the master node to become ready, launch deployments, and for other minion nodes to register and activate.

You will also need to open port 4001 in the AWS Security Group for your "masters" instances. This can be edited in the AWS EC2 Management Console. Edit the rule for TCP Ports 1-4000 from "nodes", and change the range to 1-4001.

The install for kops provides two additional components: - romana-aws: A tool that automatically configures EC2 Source-Dest-Check attributes for nodes in your Kubernetes cluster - romana-vpcrouter: A service that populates your cluster's VPC Routing tables with routes between AZs.

Please see special notes below if - you are using a non-default range for Kubernetes Service IPs - want to specify your own IP range for Pod IPs

## 2.3 Installation in other environments

Please refer to the detailed components page, and align the example configuration with the details specific to your cluster.

## 2.4 Updates coming soon

These topics still need additional explanation, instructions and guides.

- Special Notes
- Custom range for Kubernetes Service IPs
- Custom range for Pod IPs
- Running in VirtualBox
- Running in multiple subnets

CHAPTER 3

# Operations

## 3.1 Upgrading Romana on Kubernetes

Romana can be upgraded by simply updating the conatiner image used in the deployment and/or daemonsets.

```
kubectl -n kube-system set image deployment/romana-daemon romana-daemon=quay.io/
↪romana/daemon:v2.0.0
kubectl -n kube-system set image deployment/romana-listener romana-listener=quay.io/
↪romana/listener:v2.0.0

kubectl -n kube-system set image daemonset/romana-agent romana-agent=quay.io/romana/
↪agent:v2.0.0
```

Upgrading the *romana-agent* requires the additional step of changing the "update strategy" from the default *OnDelete* to *RollingUpdate*.

This is done by running

```
kubectl -n kube-system edit daemonset romana-agent
```

Then changing *OnDelete* to *RollingUpdate*.

For upgrades from preview.3 to v2.0 GA, no etcd data migration is necessary.

## 3.2 Romana Command Line Tools

Since Romana is controlled via a cloud orchestration system, once it is installed and running requires little operational oversight. However, for certain adminstrative functions a CLI is provided.

**Note:** The Romana CLI retains certain Romana v1.x elements, including commands to directly administer default (i.e. pre-defined) *tenant* and *segment* labels. An update to the CLI will soon be available that allows for dynamic label creation.

---

Romana command line tools provide a romana API reference implementation. They provide a simple command line interface to interact with Romana services.

### 3.2.1 Setting up CLI

**./romana** CLI uses a configuration file ~/.romana.yaml which contains various parameters for connecting to root service and root service port. A sample ~/.romana.yaml file looks as follows:

```
$ cat ~/.romana.yaml
#
# Default romana configuration file.
# please move it to ~/.romana.yaml
#
RootURL: "http://192.168.99.10"
RootPort: 9600
LogFile: "/var/tmp/romana.log"
Format: "table" # options are table/json
Platform: "openstack"
Verbose: false
```

### 3.2.2 Basic Usage

Once a configuration is setup (by default the romana installer will populate the ~/.romana.yaml with a valid configuration), running the *romana* command will display details about commands supported by romana.

```
Usage:
  romana [flags]
  romana [command]

Available Commands:
  host        Add, Remove or Show hosts for romana services.
  tenant      Create, Delete, Show or List Tenant Details.
  segment     Add or Remove a segment.
  policy      Add, Remove or List a policy.

Flags:
  -c, --config string     config file (default is $HOME/.romana.yaml)
  -f, --format string     enable formatting options like [json|table], etc.
  -h, --help              help for romana
  -P, --platform string   Use platforms like [openstack|kubernetes], etc.
  -p, --rootPort string   root service port, e.g. 9600
  -r, --rootURL string    root service url, e.g. http://192.168.0.1
  -v, --verbose           Verbose output.
      --version           Build and Versioning Information.
```

### 3.2.3 Host sub-commands

#### Adding a new host to romana cluster

Adding a new host to romana cluster should be done using static hosts and this feature is only avaiable here for debugging assistance.

```
romana host add [hostname][hostip][romana cidr][(optional)agent port] [flags]
```

#### Removing a host from romana cluster

```
romana host remove [hostname|hostip] [flags]
```

#### Listing hosts in a romana cluster

```
romana host list [flags]
```

#### Showing details about specific hosts in a romana cluster

```
romana host show [hostname1][hostname2]... [flags]
```

### 3.2.4 Tenant sub-commands

#### Create a new tenant in romana cluster

Creating a new tenant is only necessary on certain platforms like openstack (where the tenant has to exist previously on that platform), for platforms like kubernetes, tenants are created automatically and no command line interaction is needed in those cases.

```
romana tenant create [tenantname] [flags]
```

#### Delete a specific tenant in romana cluster

```
romana tenant delete [tenantname] [flags]
```

#### Listing tenants in a romana cluster

```
romana tenant list [flags]
```

#### Showing details about specific tenant in a romana cluster

```
romana tenant show [tenantname1][tenantname2]... [flags]
```

### 3.2.5 Segment sub-commands

#### Add a new segment to a specific tenant in romana cluster

Adding a new segment to a specific tenant is only necessary on certain platforms like openstack, for platforms like kubernetes, segments are created automatically and no command line interaction is needed in those cases.

```
romana segment add [tenantName][segmentName] [flags]
```

#### Remove a segment for a specific tenant in romana cluster

```
romana segment remove [tenantName][segmentName] [flags]
```

#### Listing all segments for given tenants in a romana cluster

```
romana segment list [tenantName][tenantName]... [flags]
```

### 3.2.6 Policy sub-commands

#### Sample Romana Policy

A sample romana policy is shown here.

#### Add a new policy to romana cluster

Adding policies to romana cluster involves them being applied to various backends like openstack VMs, Kubernetes Pods, etc for various platforms supported by romana.

```
romana policy add [policyFile] [flags]
```

Alternatively policies can be added using standard input.

```
cat policy.json | romana policy add
```

#### Remove a specific policy from romana cluster

```
romana policy remove [policyName] [flags]
Local Flags:
    -i, --policyid uint   Policy ID
```

#### Listing all policies in a romana cluster

```
romana policy list [flags]
```

CHAPTER 4

# Romana Networking

This document explains some of Romana's core networking concepts. Oftentimes, a detailed understanding of those is not required, since Romana strives to provide sensible defaults and working out-of-the-box configurations. But for custom deployments or advanced configuration, having an understanding of those concepts is helpful.

## 4.1 Terminology

The following terminology is used throughout this document:

*agent*: This is a Romana component, which runs on every node in a cluster. This agent is used to configure interfaces, routing or network policies on that host.

*CIDR*: 'Classless Inter-Domain Routing'. We use the term CIDR to refer specifically to the 'CIDR notation', which is a compact representation of an IP address and routing prefix. Most commonly, however, we use this notation to describe ranges of addresses. For example, the CIDR of `10.1.0.0/16` describes all IP addresses from `10.1.0.0` to `10.1.255.255`.

*Clos Network*: A Clos network is a structured, hierarchical *network topology* often used for data center networks. It can be described as a tree-like 'spine and leaf' architecture, with racks of servers as the bottom layer. Those servers are typically connected to a ToR ('top of rack') switch or router, which represents the next layer. An entire rack (servers plus ToR) may be considered a 'leaf' in this architecture. Core routers or switches are then used to connect those ToRs, often in a highly redundant manner, which can tolerate failure of core devices. Those core devices then form the 'spine' in the architecture. This is a simplified description and countless variations of Clos networks exist. For example, additional layers may be inserted. Or in various large core networks, and entire spine may be considered the 'leaf', depending on the scope of the discussion and network. Connections between spines and leafs may be managed by vendor specific network fabric implementation, and so on.

*endpoint*: Romana deals with the provisioning of networking for *workloads* in clusters (VMs for OpenStack, pods for Kubernetes). Due to a network-centric view, for Romana the most important aspect of a workload is the 'network endpoint', which is the IP address and network interface of the workload.

*host*: In this context, usually a server computer that is part of a cluster, such as OpenStack or Kubernetes. In this document we use the term 'host' or 'node' interchangeably.

*IPAM*: 'IP Address Manager'. A service that manages IP addresses in a network. If anyone in the network needs a new address, a request can be sent to IPAM to get the 'next' available IP address from some pre configured range. Romana's IPAM is an extremely important component since carefully chosen addresses and network prefixes are used to greatly *collapse routes* and reduces the impact of route distribution on hosts and networking equipment.

*master node*: One of the *nodes* of an OpenStack or Kubernetes cluster, which fulfills 'master' or 'controller' functions. This is typically where central components of the cluster infrastructure run. *Workloads* (VMs or pods) may or may not run on master nodes, depending on configuration.

*node*: A host that is a member of a cluster (either OpenStack or Kubernetes). In this document we use the term 'host' and 'node' interchangeably.

*policy*: Romana provides network policies to manage network traffic and allow or deny access. You can always use the Romana CLI and API to define policies. For Kubernetes clusters, Romana also implements a direct and automatic mapping of Kubernetes policies to Romana policies.

*route aggregation*: Since all *endpoint* IP addresses are fully routable, Romana needs to configure and manage the routes within the network. Great emphasis has been placed on collapsing those routes (aggregation), so that only very few routes actually need to be configured and maintained within the network infrastructure. For example, if there are two routes to adjacent *CIDRs* `10.1.0.0/25` and `10.1.0.128/25`, which point to the same target then this can be collapsed into a single route for the CIDR `10.1.0.0/24`. Romana facilitates a special *IPAM* that is aware of network topology in order to assign IP addresses to endpoints in a manner that makes it easy to automatically aggregate routes.

*network topology*: A network topology describes the layout of elements in a network. How are hosts, switches and routers connected? How can two components communicate? Who can reach whom directly and who needs to forward packets on behalf of others? Traditional network topologies are 'bus', 'star', 'ring' or 'mesh'. In modern data centers topologies can be complex and may contain some mixture of those types, often adding certain tree-like hierarchical aspects to the topology, for example in *Clos networks*.

*workload*: In OpenStack clusters this is typically a VM, while in Kubernetes clusters this is normally a pod. These workloads run on the cluster *hosts*. Each workload is represented by a network *endpoint*, consisting of a network interface that is configured on the host as well as an assigned IP address.

## 4.2 Networking

### 4.2.1 Fully routed networks without overlays

Romana does not use network overlays or tunnels. Instead, *endpoints* (VMs for OpenStack, pods for Kubernetes) get real, routable IP addresses, which are configured on cluster *hosts*. To ensure connectivity between endpoints, Romana manages routes on cluster hosts as well as network equipment or in the cloud infrastructure, as needed.

Using real, routable IP addresses has multiple advantages:

- Performance: Traffic is forwarded and processed by hosts and network equipment at full speed, no cycles are spent encapsulating packets.
- Scalability: Native, routed IP networking offers tremendous scalability, as demonstrated by the Internet itself. Romana's use of routed IP addressing for endpoints means that no time, CPU or memory intensive tunnels or other encapsulation needs to be managed or maintained and that network equipment can run at optimal efficiency.
- Visibility: Packet traces show the real IP addresses, allowing for easier trouble shooting and traffic management.

### 4.2.2 Romana address blocks

To increase scalability and reduce resource requirements, Romana allocates *endpoint* addresses not individually, but in blocks of addresses. Each block can be expressed in the form of a *CIDR*. For example, a typical address block may have 4 bits (a `/28` CIDR), and therefore could contain up to 16 different IP addresses. *Routing is managed* based on those blocks. Only one network route is required for all addresses within this block. Therefore, address blocks are an important contribution to *route aggregation*.

Let's look at an example to illustrate:

Assume Romana has been configured to use addresses out of the `10.1.0.0/16` address range and to use `/28` blocks. Now assume a first workload needs to be started. The cluster scheduler decided to place this workload on host A of the cluster.

Romana's IPAM realizes that no block is in use yet on host A, allocates a block and 'assigns' it to host A. For example, it may choose to use block `10.1.9.16/28`. As this assignment is made, the Romana agent on the cluster hosts *may create routes* to this block's address range, which point at host A. The address `10.1.9.17` (contained in that address block) could be chosen by IPAM and is returned as the result of the original address request. Therefore, the first endpoint gets address `10.1.9.17`.

Now a second endpoint needs to be brought up. The scheduler chose host B. IPAM finds that no block is present on host B yet, chooses one (maybe `10.1.9.32/28`) and returns an IP address from that block. For example `10.1.9.33`.

The two endpoints (`10.1.9.17` on host A and `10.1.9.33` on host B) can communicate with each other, because Romana automatically setup routing for those address blocks.

If now a third endpoint needs to be brought up, and it is again scheduled to host A, then IPAM detects that there is an address block already on host A, but it is not fully used, yet. Therefore, it returns a free address from that block, for example `10.1.9.18`. Importantly, no new block allocation was necessary in that case, an no additional routes had to be configured. This image illustrates the state at this point:

As a result, the need to update routes on hosts or in the network infrastructure is greatly reduced. The larger the address blocks, the less often routes have to be configured or updated.

Choosing the right address block size is a tradeoff between the number of routes on one hand, as well as potentially wasted IP addresses on the other: If the block size was chosen too large then some IP addresses may never be used. For example, imagine a block size of /24. The block may contain up to 256 addresses. If on a particular host you never run that many workloads then some of those addresses may be wasted, since they are not available on other hosts.

If a block size is chosen too small then for a cluster with many endpoints Romana has to create a lot of routes (either on the hosts or the network equipment). Romana provides many features to reduce the number of routes and route updates in the network and therefore - for most cases - we recommend address block sizes of at least 4 or 5 bits.

An address block, while in use, is tied to a specific host. When workloads are stopped and the last address within a block is released, the block itself goes back into Romana IPAM's free pool. When it is used the next time, it may be allocated to a different host.

### 4.2.3 Route management

Depending on the *network's topology* Romana creates and manages routes for *address blocks* by a number of different means.

In most cases, the *Romana agents* on the *cluster hosts* create routes to address blocks on other cluster hosts, at least for those hosts that are on the same L2 segment. This is often the case if the ToR acts as a switch for the hosts in the rack and is sometimes described as 'L2-to-host'. The following image illustrates this network configuration:

Some networks are designed for 'L3-to-host', meaning that hosts do not share an L2 segment. In this case, no routes need to be configured on the hosts at all. Routes to address blocks are installed on the ToR instead. Traffic will simply

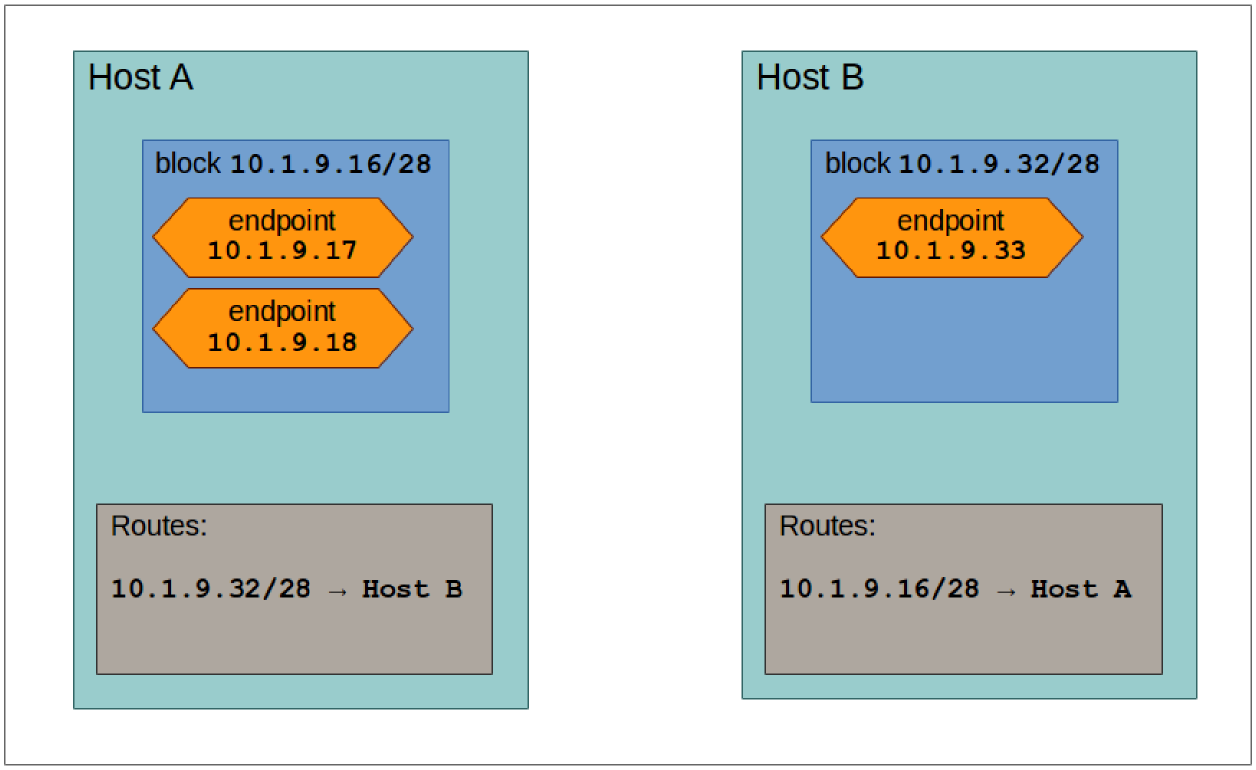

Fig. 1: State in a the cluster after third endpoint was created

use the default route to the ToR switch where it will forward to the propher endpoint. The following image shows where routes are created in an L3-to-host configuration:

Routes can be advertized to the ToR using either BGP or OSPF route distribution protocols.

Romana is provided with topological information about the network in which it is deployed as a configuration parameter. It then uses this information to maintain aggregated routes <#term_aggregation>‘__ which reduces the number of routes that need to be created and updated. In many cases all endpoints can be reachable with very small numbers of routes and few, if any, route updates required.

## 4.3 Topology

### 4.3.1 Prefix groups

*Prefix groups* are one of the key ideas behind Romana’s *IPAM*. With this concept, IP addresses for *endpoints* are chosen from the same *CIDR* if they are created in ‘close proximity’.

For example, assume you run a cluster in a data center network, consisting of multiple racks full of servers. Romana IPAM may consider all the hosts within a rack to be part of the same prefix group. This means that all *address blocks* - and therefore all endpoint IP addresses - assigned to those hosts will share the same address prefix. This then means that the ToRs (top of rack) switches in the data center only need to know a single route to be able to send traffic to all the endpoints within a rack: With this topology aware IPAM, Romana is able to drastically collapse the routing table, reducing the memory requirements, CPU load and network load of the network infrastructure.

Prefix groups also allows more specific route filtering between routers which can prevent route injection attacks.

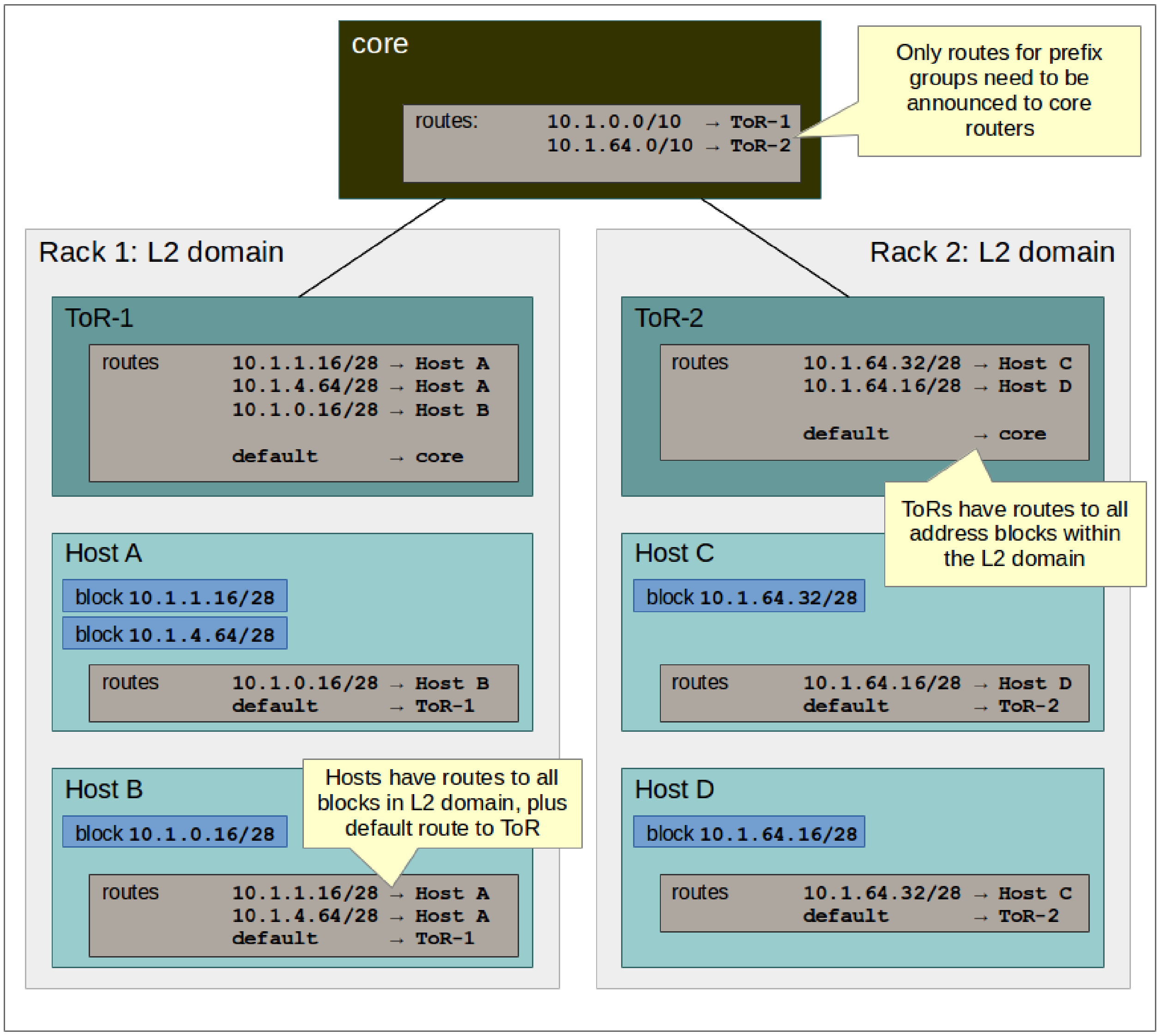
core
routes: 10.1.0.0/10 → ToR-1
10.1.64.0/10 → ToR-2
Only routes for prefix groups need to be announced to core routers
Rack 1: L2 domain
Rack 2: L2 domain
ToR-1
routes 10.1.1.16/28 → Host A
10.1.4.64/28 → Host A
10.1.0.16/28 → Host B
default → core
ToR-2
routes 10.1.64.32/28 → Host C
10.1.64.16/28 → Host D
default → core
ToRs have routes to all address blocks within the L2 domain
Host A
block 10.1.1.16/28
block 10.1.4.64/28
routes 10.1.0.16/28 → Host B
default → ToR-1
Host C
block 10.1.64.32/28
routes 10.1.64.16/28 → Host D
default → ToR-2
Host B
block 10.1.0.16/28
Hosts have routes to all blocks in L2 domain, plus default route to ToR
routes 10.1.1.16/28 → Host A
10.1.4.64/28 → Host A
default → ToR-1
Host D
block 10.1.64.16/28
routes 10.1.64.32/28 → Host C
default → ToR-2
Figure 2: Romana blocks and routes in an 'L2-to-the-host' data center

Fig. 2: Routes in an L2-to-host data center

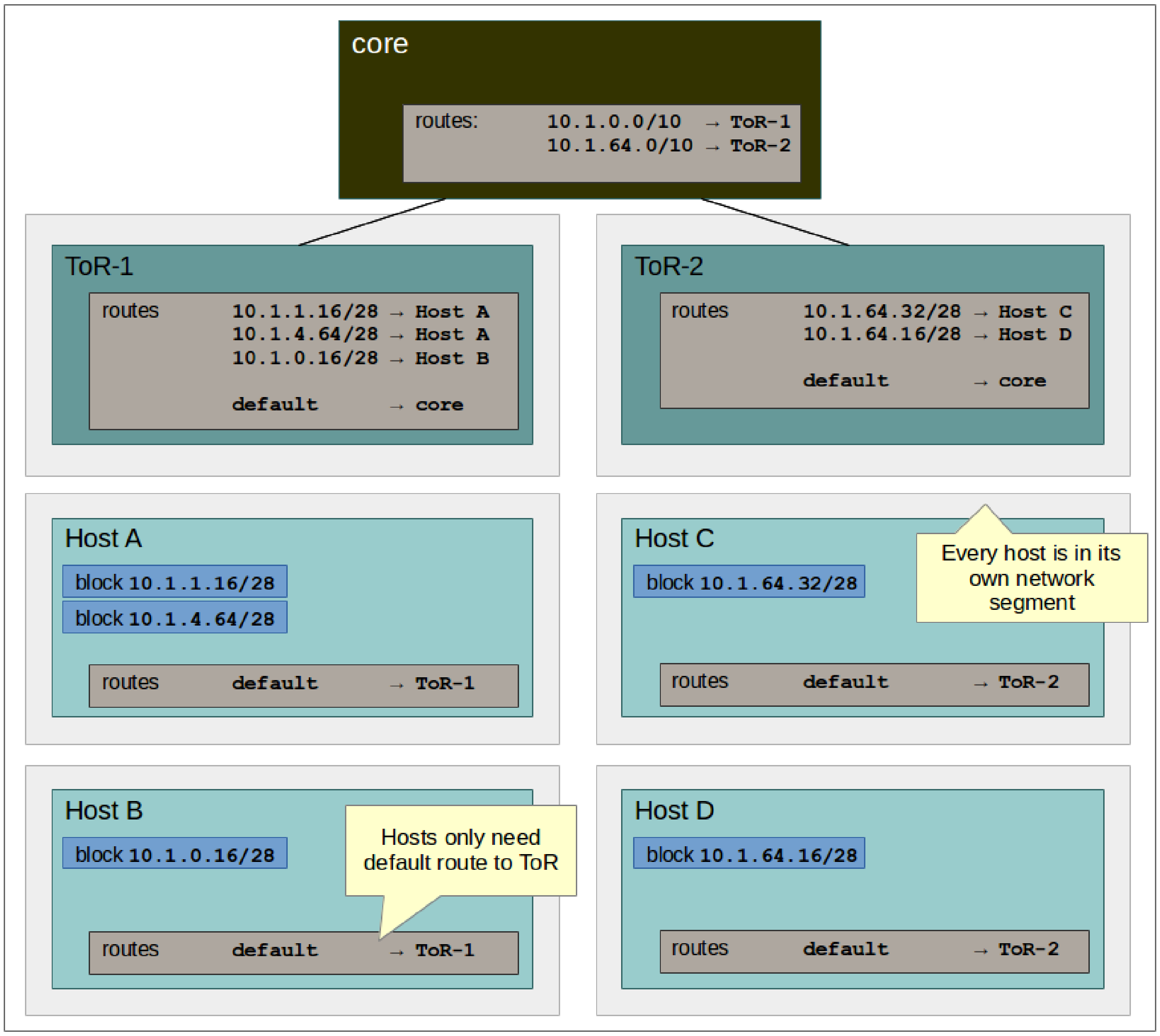

Fig. 3: Routes in an L3-to-host data center

Let's look at an example in more detail.

Assume your data center consists of four racks. Each rack has a ToR leaf, connected to a pair of spine core routers.

Assume further that the overall address range for Romana is `10.1.0.0/16`.

These four racks are specified in a *topology map*: a configuration that describes the *network topology* and which is provided to Romana as input. Romana then takes this information and automatically carves up the overall CIDR available into four sub-ranges: `10.1.0.0/18`, `10.1.64.0/18`, `10.1.128.0/18` and `10.1.192.0/18`. It then assigns these sub-ranges as a prefix groups to the ToR and organizes the hosts in each rack to get addresses from within the prefix group. For example, `10.1.0.0/18` may be assigned to rack 1, `10.1.64.0/18` to rack 2, and so on.

Then, if the cluster scheduler wishes to bring up a workload on any host in rack 1, Romana IPAM will make sure that the address block used for this endpoint will be fully contained in the `10.1.0.0/18` CIDR. For example, the address block may have the CIDR `10.1.0.8/28`.

Likewise, if an address block is needed on any host in rack 2, it will have a CIDR that's contained within the second prefix group's CIDR. For example, `10.1.64.8/28`.

As a result, to send outgoing packets to endpoints in other racks, the spine routers only need to have four routes: One route for each prefix-group's CIDR to the ToR for that prefix-group / rack. These routes do not even require updating.

Note that every environment is different. Romana provides for a great deal of flexibility to organize hosts into prefix groups and how to configure the announcement of routes. Prefix groups are not only important in data centers, but also in clusters that are running on cloud infrastructure. Where and how routes are announced and created may differ depending on the environment. Romana supports a number of options.

### 4.3.2 Topology map

A *topology map* is one of the configuration parameters for Romana and is the basis on which Romana *IPAM* calculates *CIDRs* for *prefix groups*. The topology map is a representation of certain aspects of the actual *network topology*.

Here are a few simplified examples:

#### Example 1: Flat network, single prefix group

In this example, any host that is added to the cluster will be automatically assigned to the single prefix group we have defined here.

```
{
    ...
    "map" : [
        {
            "name"   : "all-hosts",
            "groups" : []
        }
    ]
    ...
}
```

The CIDR of the prefix groups will be the entire CIDR given to Romana to work with.

#### Example 2: Data center with four racks

Here, we define a topology with four prefix group, one for each rack in our data center.

Note the 'assignment' specifier. This matches any tags assigned to *cluster hosts*. Therefore, as cluster nodes are added, the operator should ensure that tags with those values are specified for each host. Both OpenStack as well as Kubernetes offer the option to tag hosts as they are added to the cluster. In some cloud environments, hosts are automatically added with a region or zone identifier, which can then be used in the same manner.

```
{
    ...
    "map" : [
        {
            "name"       : "rack-1",
            "assignment" : { "my-location-tag" : "rack-1" },
            "groups"     : []
        },
        {
            "name"       : "rack-2",
            "assignment" : { "my-location-tag" : "rack-2" },
            "groups"     : []
        },
        {
            "name"       : "rack-3",
            "assignment" : { "my-location-tag" : "rack-3" },
            "groups"     : []
        },
        {
            "name"       : "rack-4",
            "assignment" : { "my-location-tag" : "rack-4" },
            "groups"     : []
        },
    ]
    ...
}
```

In this example, Romana's entire address range is automatically split into four CIDRs and each of those CIDRs is assigned to one prefix group. This means that all *endpoints* in a given rack will share the same address prefix, which allows for the complete aggregation of all routes for the endpoints in that rack.

More comples examples for a number of real world topology maps are available in Custom Topologies section of 'Advanced Topics'

# CHAPTER 5

# Romana Components

An installation of Romana on Kubernetes has a number of essential components, and some add-on components for specific cloud providers. These are currently available for AWS, and components for other cloud providers will be developed in the future.

Details of each component and example YAML configurations are provided below.

## 5.1 Essential Components

### 5.1.1 romana-etcd

A Romana installation requires access to `etcd` for storing information about hosts, routing, IP addresses and other essential configuration and state. This can be the same etcd storage used by Kubernetes itself, dedicated etcd storage for Romana, or a standalone pod.

#### Expose Kubernetes etcd

If you are using the Kubernetes etcd storage for Romana, then it is exposed as a service. See the example etcd-service YAML file. To match this with a custom environment, you need to ensure

- The `clusterIP` isspecified and a valid value for your cluster's `--service-cluster-ip-range`. The value for this range can be found in the configuration for your `kube-apiserver`.
- The `targetPort` must match the port used by clients to connect to etcd. You will find this value in the environment variable `ETCD_LISTEN_CLIENT_URLS` or the command-line option `--listen-client-urls` for etcd.
- The `selector` lists labels that must match your etcd pods. Please ensure your etcd pods have a distinct label and that the `selector` matches that label.

#### Dedicated etcd storage

You can deploy your own etcd instance or cluster within Kubernetes and make Romana use that instead of the Kubernetes etcd. It's highly recommended to expose that dedicated etcd storage as a service. See the section above for details.

#### Standalone pod

In simplified environments with a single master node, for demonstration or experimentation, you can create a standalone etcd instance for Romana to use. See the example etcd-standalone YAML file. This is not recommended for production purposes because it is not fault-tolerant - losing the master node means losing critical data and state for both Kubernetes and Romana.

The example contains two parts that need to be aligned:

- the `romana-etcd` Service
- the `romana-etcd` Deployment

The following details must be modified to match your cluster's settings:

- Service IP

  The Service IP for `romana-etcd` needs to be a valid value for your cluster's `--service-cluster-ip-range` CIDR, which is configured in your kube-apiserver.

  The value needs to be specified in the `romana-etcd` service for `clusterIP`, and also in the `romana-etcd` deployment template for the `--advertise-client-urls` option.

- Port

  The port for `romana-etcd` needs to be specified in the `romana-etcd` service for `port`, in the `romana-etcd` deployment template for the `--listen-client-urls` option, and in the `livenessProbe` for the `port`.

- Target Port

  The Target Port for `romana-etcd` needs to be specified in the `romana-etcd` service for `targetPort`, and in the `romana-etcd` deployment template for the `--advertise-client-urls` option.

- Labels

  The same labels should be used in the `romana-etcd` service for `selector` and in the `romana-etcd` deployment template for `labels` in the metadata.

- Placement

  The pod should be forced to run on a specific master node. If your master has a unique `node-role` label, then that can be used in the `romana-etcd` deployment template for the `nodeSelector`. Otherwise, the `nodeSelector` should be updated to match the key and value for the master node's `kubernetes.io/hostname`

  If your master node is *tainted* to prevent pods being scheduled there, the `romana-etcd` deployment template should include the matching `toleration` to permit this pod.

### 5.1.2 romana-daemon

The `romana-daemon` service is a central service used by other Romana components and provides an API for queries and changes. See the example romana-daemon YAML file.

The example contains two parts that need to be aligned:

- the `romana-daemon` Service
- the `romana-daemon` Deployment

The following details must be modified to match your cluster's settings:

- Service IP

  The Service IP for `romana-daemon` needs to be a valid value for your cluster's `--service-cluster-ip-range` CIDR, which is configured in your kube-apiserver.

  The value needs to be specified in the `romana-daemon` service for `clusterIP`.

- Placement

  The pod should be forced to run on a master node. If your master has a unique `node-role` label, then that can be used in the `romana-daemon` deployment template for the `nodeSelector`. Otherwise, the `nodeSelector` should be updated to match the key and value for the master node's `kubernetes.io/hostname`

  If your master node is *tainted* to prevent pods being scheduled there, the `romana-daemon` deployment template should include the matching `toleration` to permit this pod.

- Cloud Provider Integration

  If your Kubernetes cluster is running in AWS and configured with `--cloud=aws`, then you should provide that option to the romana-daemon.

  This is done by uncommenting the `args` section and `--cloud` option in the `romana-daemon` deployment template.

`yaml args:  - --cloud=aws`

- Initial Network Configuration

To complete the configuration of Romana, a network topology needs to be configured. There are some built-in network topologies that will be used if possible, but in custom environments, this will need to be provided by the user.

A built-in topology will be used if the `--cloud=aws` option was specified, or if the default Kubernetes Service IP is detected for `kops` or `kubeadm` (100.64.0.1 for kops, 10.96.0.1 for kubeadm).

A user-defined network topology can be provided by - loading the network topology file into a configmap using kubectl

> `` `bash kubectl -n kube-system create configmap romana-network-conf --from-file=custom-network.json ` ``

- mounting the configmap into the romana-daemon pod

`yaml volumeMounts:  - name:  romana-config-volume mountPath:  /etc/romana/network volumes:  - name:  romana-config-volume configMap:  name: romana-network-conf`

- specifying the path to that network topology file in the romana-daemon pod arguments

`yaml args:  - --initial-network=/etc/romana/network/custom-network.json`

The path is a combination of the `mountPath` (eg: `/etc/romana/network`) and the filename inside the configmap (eg: `custom-network.json`).

See the example romana-daemon-custom-network YAML file.

- Network CIDR Overrides

When using a built-in topology, the configuration specifies the CIDR that will be used for allocating IP addresses to pods.

This value can be changed by specifying the `--network-cidr-overrides` option in the `romana-daemon` deployment template

`yaml args:  - --network-cidr-overrides=romana-network=100.96.0.0/11`

The value for the CIDR should not overlap with any existing physical network ranges, or the Kubernetes `service-cluster-ip-range`.

### 5.1.3 romana-listener

The `romana-listener` service is a background service that listens for events from the Kubernetes API Server and updates configuration in Romana. See the example romana-listener YAML file.

The example contains four parts: - the `romana-listener` ClusterRole - the `romana-listener` ServiceAccount - the `romana-listener` ClusterRoleBinding - the `romana-listener` Deployment

The following details must be modified to match your cluster's settings:

- Placement

  The pod should be forced to run on a master node. If your master has a unique `node-role` label, then that can be used in the `romana-listener` deployment template for the `nodeSelector`. Otherwise, the `nodeSelector` should be updated to match the key and value for the master node's `kubernetes.io/hostname`

  If your master node is *tainted* to prevent pods being scheduled there, the `romana-listener` deployment template should include the matching `toleration` to permit this pod.

### 5.1.4 romana-agent

The `romana-agent` component is a local agent than runs on all Kubernetes nodes. It installs the CNI tools and configuration necessary to integrate Kubernetes CNI mechanics with Romana, and manages node-specific configuration for routing and policy. See the example romana-agent YAML file.

The example contains four parts:

- the `romana-agent` ClusterRole
- the `romana-agent` ServiceAccount
- the `romana-agent` ClusterRoleBinding
- the `romana-agent` DaemonSet

The following details must be modified to match your cluster's settings:

- Service Cluster IP Range

  The Service Cluster IP Range for your Kubernetes cluster needs to be passed to the `romana-agent`, matching the value that is configured in your kube-apiserver. A default value will be used if the default Kubernetes Service IP is detected for `kops` or `kubeadm` (100.64.0.1 for kops, 10.96.0.1 for kubeadm).

  This value can be changed by specifying the `--service-cluster-ip-range` option in the `romana-daemon` deployment template

`yaml args:  - --service-cluster-ip-range=100.64.0.0/13`

- Placement

The pod should be forced to run on all Kubernetes nodes. If your master node(s) are *tainted* to prevent pods being scheduled there, the `romana-agent` daemonset template should include the matching `toleration` to permit this pod.

## 5.2 AWS Add-on Components

For operation in AWS two additional components are installed.

### 5.2.1 romana-aws

The `romana-aws` service listens for node information from the Kubernetes API Server and disables the Source-Dest-Check attribute of the EC2 instances to allow pods to communicate between nodes. See the example romana-aws YAML file.

The following details must be modified to match your cluster's settings:

- Placement

  The pod should be forced to run on a master node. If your master has a unique `node-role` label, then that can be used in the `romana-aws` deployment template for the `nodeSelector`. Otherwise, the `nodeSelector` should be updated to match the key and value for the master node's `kubernetes.io/hostname`

  If your master node is *tainted* to prevent pods being scheduled there, the `romana-aws` deployment template should include the matching `toleration` to permit this pod.

- IAM Permissions

  The IAM role for your master node(s) needs to include the permission to modify EC2 Instance Attributes.

### 5.2.2 romana-vpcrouter

The `romana-vpcrouter` service is responsible for creating and maintaining routes between Availability Zones and Subnets for a Kubernetes cluster in AWS. It combines node state information from Kubernetes, AWS and internal monitoring, and route assignments from Romana, and uses this to add and modify routes in the VPC Routing Tables.

The following details must be modified to match your cluster's settings:

- `romana-etcd` Service IP and Port

  The Service IP and Target Port for `romana-etcd` need to be specified in the `romana-vpcrouter` deployment template as values for the `--etcd_addr` and `--etcd_port` options.

- Placement The pod should be forced to run on a master node. If your master has a unique `node-role` label, then that can be used in the `romana-vpcrouter` deployment template for the `nodeSelector`. Otherwise, the `nodeSelector` should be updated to match the key and value for the master node's `kubernetes.io/hostname`

  If your master node is *tainted* to prevent pods being scheduled there, the `romana-vpcrouter` deployment template should include the matching `toleration` to permit this pod.

- IAM Permissions

  The IAM role for your master node(s) needs to include the permission to describe EC2 Resources, list and modify VPCs, and list and modify RouteTables.

- Security Groups

The vpcrouter component performs active liveness checks on cluster nodes. By default, it uses ICMPecho ("ping") requests for this purpose. Therefore, please ensure that your security group ruless allow for cluster nodes to exchange those messages.

CHAPTER 6

# Network Policies

Romana allows the fine grained control and management of network traffic via network policies. The Romana network policies format was inspired by the Kubernetes network policy specification. However, Romana policies can be applied in Kubernetes as well as OpenStack environments. Furthermore, Romana extends the policies with additional features, such as the ability to control network traffic not only for containers or VMs, but also for bare metal servers.

## 6.1 Overview

Network policies are defined as small JSON snippets, specifying match characteristics for network traffic. Essentially, network policies firewall rules definitions. Details and examples will be given below.

These policy definitions are sent to the Romana Policy service using this service's RESTful API. The service validates those policies and forwards them to the Romana agent on each host of the cluster. There, the policies are translated to iptables rules, which are then applied to the kernel.

## 6.2 Tools and integration

After installing an OpenStack or Kubernetes cluster with Romana, the `romana` command line tool can be used to specify and list policies. However, Romana provides a specific integration for Kubernetes. This allows the operator to use standard Kubernetes policies and policy APIs, should they wish to do so. Romana picks up those Kubernetes policies, seamlessly translates them to Romana policies and then applies them as necessary.

For OpenStack, or if policies need to be applied to bare metal servers, the Romana Policy API or command line tools are used directly.

### 6.2.1 Policy definition format

Each Romana network policy document contains a single top-level element (`securitypolicies`), which itself is a list of individual policies. A policy contains the following top-level elements:

- **name:** The name of the policy. You can refer to policies by name or an automatically generated unique ID. Oftentimes names are much easier to remember. Therefore, it is useful to make this a short, descriptive and - if possible - unique ID.
- **description:** A line of text, which can serve as human readable documentation for this policy.
- **direction:** Determines whether the policy applies packets that are incoming (ingress) to the endpoint or outgoing (egress) from the endpoint. Currently, the only permissible value for this field is `ingress`. This means that the policy rules describe traffic travelling TO the specified (see `applied_to`) target.
- **applied_to:** A list of specifiers, defining to whom the rules are applied. Typically a tenant/segment combo or a CIDR.
- **peers:** A list of specifiers, defining the 'other side' of the traffic. In case of ingress traffic, this would be the originator of the packets. The peer may be defined as "any", which serves as a wildcard.
- **rules:** A list of traffic type specifications, usually consisting of protocol and ports.

```
{
    "securitypolicies": [{
        "name":        <policy-name>,
        "description": <policy-description>,
        "direction":   "ingress",
        "applied_to":  [<applied-spec-1>, <applied-spec-2>, ...],
        "peers":       [<peer-spec-1>, <peer-spec-2>, ...],
        "rules":       [<traffic-spec-1>, <traffic-spec-2>, ...]
    }]
}
```

Example:

```
{
    "securitypolicies": [{
        "name": "policy1",
        "description": "Opening SSH, HTTP and HTTPS ports as well as ICMP",
        "direction": "ingress",
        "applied_to": [{
            "tenant": "admin",
            "segment": "default"
        }],
        "peers": [{
            "peer": "any"
        }],
        "rules": [
            {
                "protocol": "tcp",
                "ports": [22, 80, 443]
            },
            {
                "protocol": "icmp"
            }
        ]
    }]
}
```

CHAPTER 7

# Network Topology

To make Romana aware of important details of your network, it is configured using a *network topology* configuration. This is a JSON formatted file that describes the network(s) that will be used for Kubernetes pods, and links them to the physical network that is hosting them.

If you are deploying your Kubernetes cluster with a recognized tool such as kops or kubeadm, your installation should use an existing predefined topology. For other environments including customized installations and baremetal deployments, the information about your networks will need to be provided.

## 7.1 Network Topology Configuration Format

### 7.1.1 Network Topology JSON

```
{
    "networks": [ Network Definition, ... ]
    "topologies": [ Topology Mapping, ... ]
}
```

- `networks` (required)

A list of *Network Definition* objects. These describe the names of the networks and the CIDR that pod addresses will be allocated from.

- `topologies` (required)

A list of *Topology Mapping* objects. These link the network definitions and the topology of hosts within the cluster.

### 7.1.2 Network Definition JSON

```
{
    "name": String,
```

(continues on next page)

(continued from previous page)

```
    "cidr": IPv4 CIDR,
    "block_mask": Number
}
```

- `name` (required)

The name for this network. Each name must be unique. This name is used to link network definitions and topology mappings.

- `cidr` (required)

The IPv4 CIDR for pods created within this network. Each CIDR must be unique, not overlapping with other values, and also not overlapping your cluster's `service-cluster-ip-range`.

- `block_mask` (required)

The mask applied to address blocks. This must be longer than the mask used for the CIDR, with a maximum value of 32. It implicitly defines the number of addresses per block, eg: a value of /29 means the address block contains 8 addresses.

### 7.1.3 Topology Mapping JSON

```
{
    "networks": [ String, ... ],
    "map": [ Host Group, ... ]
}
```

- `networks` (required)

A list of network names. All values must match the name of an object from the top-level `networks` list.

- `map` (required)

A list of *Host Group* objects. This is a topology map for the list of networks.

### 7.1.4 Host Group JSON

```
{
    "name": String,
    "hosts": [ Host Definition, ... ],
    "groups": [ Host Group, ... ],
    "assignment": { String: String, ... }
}
```

- `name` (optional)

A descriptive name for this mapping item.

- `hosts` (conditional)

A list of *Host Definition* objects. Only one of "hosts" and "groups" can be specified.

- `groups` (conditional)

A list of Host Group objects. Only one of "hosts" and "groups" should be specified. This allows for nesting the definition of groups to match your topology at each level, eg: spine and leaf. Nested groups are treated as prefix groups for IP addressing and routing..

An empty list may be specified. This indicates the lowest level of grouping, but without defining hosts.

- `assignment` (conditional)

A list of key-value pairs that correspond to Kubernetes `node` labels. These are used to assign Kubernetes nodes to a specific Host Group. In networks with multiple subnets, it is recommended that your Kubernetes nodes use the appropriate `failure-domain` lables, and matching those labels and values with the `assignment` in your topology config.

## 7.2 Host Definition JSON

```
{
    "name": String,
    "ip", String
}
```

- `name` (required)

The name of the host. Each name must be unique. This name must match the node name registered in Kubernetes.

- `ip` (required)

The IP address of the host. Each IP must be unique. This address must match the node address registered in Kubernetes.

### 7.2.1 Examples

- Network topology used for kubeadm installations

This example defines a single network named `romana-network`, and maps to a topology containing 8 `host-groups`. The empty groups are used as placeholders, and Kubernetes nodes will be assigned to the host-groups with round-robin placement.

- Network topology used for kops in us-west-1 region

This example defined a single network named `romana-network`, and contains a host-group for each Availability Zone (AZ) within the us-west-1 region. Inside each AZ host-group, there are 8 sub-groups with `assignment` labels specific to that AZ. Kubernetes nodes will be assigned to one of those sub-groups based on round-robin placement after matching the `assignment` labels.

CHAPTER 8

# Advanced Topics

## 8.1 Custom Topologies

Romana uses an advanced, *topology aware IPAM* module in order to assign IP addresses to endpoints (pods or VMs). The topology awareness of Romana's IPAM allows for the endpoint IPs to align with the topology of your network. This in turn makes it possible for Romana to effectively aggregate routes. This has many operational and security advantages:

- Ability to use native L3 routing, allowing network equipment to work at its best
- Greatly reduced number of routes in your networking hardware
- Stable routing configurations, less route updates required
- No "leaking" of endpoint routes into the networking fabric

Key to Romana's topology aware IPAM is the *topology configuration*. In this configuration you model the underlying network topology for your cluster.

### 8.1.1 Terminology

Some useful terminology:

- **Network**: Romana's IPAM chooses endpoint IP addresses from one or more address ranges (CIDRs). Each of those is called a "network" in the Romana topology configuration.
- **Address block**: Romana manages IP addresses in small blocks. Usually these blocks may contain 8, 16 or 32 addresses. If an IP address is needed on a host, Romana assigns one of those blocks there, then uses up the block addresses for any further endpoints on the host before assigning a new block. Block sizes are specified as network mask lengths, such as "29" (which would mean a `/29` CIDR for the block). You see this parameter in the topology configuration. It effects some networking internals, such as the number of routes created on hosts or ToRs. For the most part you don't need to worry about it and can just leave it at "29".
- **Tenant**: This may be an OpenStack tenant, or a Kubernetes namespace.

- **Group**: This is a key concept of Romana's IPAM. All hosts within a group will use endpoint addresses that share the same network prefix. That's why Romana's "groups" are also called "prefix groups". This is an important consideration for topology aware addressing and route aggregation.
- **Prefix group**: See "group".

## 8.1.2 Examples

To make it easy for you to get started we have put together this page with examples for common configurations. The configurations are specified in JSON. To explain individual lines, we have added occasional comments (starting with '#'). Since JSON does not natively support comments, you would need to strip out those before using any of these sample config files.

### Single, flat network

Use this configuration if you have hosts on a single network segment: All hosts can reach each other directly, no router is needed to forward packets. Another example may be hosts in a single AWS subnet.

Note that in the configuration we usually don't list the actual hosts. As nodes/hosts are added to a cluster, Romana selects the 'group' to which the host will be assigned automatically.

```
{
    "networks": [                           # 'networks' or CIDRs from which Romana␣
↪chooses endpoint addresses
        {
            "name"      : "my-network",     # each network needs a unique name...
            "cidr"      : "10.111.0.0/16",  # ... and a CIDR.
            "block_mask": 29                # size of address blocks for this network,
↪ safe to leave at "/29"
        }
    ],
    "topologies": [                         # list of topologies Romana knows about,␣
↪just need one here
        {
            "networks": [                   # specify the networks to which this␣
↪topology applies
                "my-network"
            ],
            "map": [                        # model the network's prefix groups
                {                           # if only one group is specified, it will␣
↪use entire network CIDR
                    "groups": []            # just one group, all hosts will be added␣
↪here
                }
            ]
        }
    ]
}
```

### Single, flat network with host-specific prefixes

Same as above, but this time we want each host to have its own 'prefix group': All endpoints on a host should share the same prefix. This is useful if you wish to manually set routes in other parts of the network, so that traffic to pods can be delivered to the correct host.

Note that Romana automatically calculates prefixes for each prefix group: The available overall address space is carved up based on the number of groups. The example below shows this in the comments.

When a host is added to a cluster, Romana assigns hosts to (prefix) groups in a round-robin sort of fashion. Therefore, if the number of defined groups is at least as high as the number of hosts in your cluster, each host will live in its own prefix group.

```
{
    "networks": [
        {
            "name"      : "my-network",
            "cidr"      : "10.111.0.0/16",
            "block_mask": 29
        }
    ],
    "topologies": [
        {
            "networks": [ "my-network" ],
            "map": [                            # add at least as many groups as you␣
↪will have hosts
                { "groups": [] },               # endpoints get addresses from 10.111.0.
↪0/18
                { "groups": [] },               # endpoints get addresses from 10.111.64.
↪0/18
                { "groups": [] },               # endpoints get addresses from 10.111.
↪128.0/18
                { "groups": [] }                # endpoints get addresses from 10.111.
↪192.0/18
            ]
        }
    ]
}
```

### Using multiple networks

Sometimes you may have multiple, smaller address ranges available for your pod or VM addresses. Romana can seamlessly use all of them. We show this using the single, flat network topology from the first example.

```
{
    "networks": [
        {
            "name"      : "net-1",
            "cidr"      : "10.111.0.0/16",
            "block_mask": 29
        },
        {
            "name"      : "net-2",             # unique names for each network
            "cidr"      : "192.168.3.0/24",    # can be non-contiguous CIDR ranges
            "block_mask": 31                   # each network can have different␣
↪block size
        }
    ],
    "topologies": [
        {
            "networks": [ "net-1", "net-2" ],   # list all networks that apply to the␣
↪topology
            "map": [
```

(continues on next page)

(continued from previous page)

```
                { "groups": [] }                # endpoints get addresses from both␣
↪networks
            ]
        }
    ]
}
```

## Using multiple topologies

It is possible to define multiple topologies, which are handled by Romana at the same time. The following example shows this. We have a total of three networks. One topology (all hosts in the same prefix group) is used for two of the networks. A third network is used by a topology, which gives each host its own prefix group (assuming the cluster does not have more than four nodes).

```
{
    "networks": [
        {
            "name"      : "net-1",
            "cidr"      : "10.111.0.0/16",
            "block_mask": 29
        },
        {
            "name"      : "net-2",
            "cidr"      : "10.222.0.0/16",
            "block_mask": 28
        },
        {
            "name"      : "net-3",
            "cidr"      : "172.16.0.0/16",
            "block_mask": 30
        }
    ],
    "topologies": [
        {
            "networks": [ "net-1", "net-2" ],
            "map": [
                { "groups": [] }                # endpoints get addresses from 10.111.
↪0.0/16 and 10.222.0.0/16
            ]
        },
        {
            "networks": [ "net-3" ],
            "map": [
                { "groups": [] },               # endpoints get addresses from 172.16.
↪0.0/18
                { "groups": [] },               # endpoints get addresses from 172.16.
↪64.0/18
                { "groups": [] },               # endpoints get addresses from 172.16.
↪128.0/18
                { "groups": [] }                # endpoints get addresses from 172.16.
↪192.0/18
            ]
        }
    ]
}
```

### Restricting tenants to networks

Romana can ensure that tenants are given addresses from specific address ranges. This allows separation of traffic in the network, using traditional CIDR based filtering and security policies.

This is accomplished via a new element: A `tenants` spec can be provided with each network definition.

Note that Romana does NOT influence the placement of new pods/VMs. This is done by the environment (Kubernetes or OpenStack) independently of Romana. Therefore, unless you have specified particular tenant-specific placement options in the environment, it is usually a good idea to re-use the same topology - or at least use a topology for all cluster hosts - for each tenant.

```
{
    "networks": [
        {
            "name"      : "production",
            "cidr"      : "10.111.0.0/16",
            "block_mask": 29,
            "tenants"   : [ "web", "app", "db" ]
        },
        {
            "name"      : "test",
            "cidr"      : "10.222.0.0/16",
            "block_mask": 32,
            "tenants"   : [ "qa", "integration" ]
        }
    ],
    "topologies": [
        {
            "networks": [ "production", "test" ],
            "map": [
                { "groups": [] }
            ]
        }
    ]
}
```

### Deployment in a multi-rack data center

The topology file is used to model your network. Let's say you wish to deploy a cluster across four racks in your data center. Let's assume each rack has a ToR and that ToRs can communicate with each other. Under each ToR (in each rack) there are multiple hosts.

As nodes/hosts are added to your cluster, you should provide labels in the meta data of each host, which can assist Romana in placing the host in the correct, rack-specific prefix group. Both Kubernetes and OpenStack allow you to define labels for nodes. You can choose whatever label names and values you wish, just make sure they express the rack of the host and are identical in the environment (Kubernetes or OpenStack) as well as in the Romana topology configuration.

In this example, we use `rack` as the label. We introduce a new element to the Romana topology configuration: The `assignment` spec, which can be part of each group definition.

Note that such a multi-rack deployment would usually also involve the installation of the *Romana route publisher*, so that ToRs can be configured with the block routes to the hosts in the rack.

```
{
    "networks": [
```

(continues on next page)

(continued from previous page)

```
        {
            "name"      : "my-network",
            "cidr"      : "10.111.0.0/16",
            "block_mask": 29
        }
    ],
    "topologies": [
        {
            "networks": [ "my-network" ],
            "map": [
                {
                    "assignment": { "rack": "rack-1" },   # all nodes with label
↪'rack == rack-1'...
                    "groups"    : []                      # ... are assigned by␣
↪Romana to this group
                },
                {
                    "assignment": { "rack": "rack-2" },
                    "groups"    : []
                },
                {
                    "assignment": { "rack": "rack-3" },
                    "groups"    : []
                },
                {
                    "assignment": { "rack": "rack-4" },
                    "groups"    : []
                },
            ]
        }
    ]
}
```

### Deployment in a multi-zone, multi-rack data center

Larger clusters may be spread over multiple data centers, or multiple spines in the data center. Romana can manage multi-hierarchy prefix groups, so that the routes across the DCs or spines can be aggregated into a single route.

The following example shows a cluster deployed across two "zones" (DCs or spines), with four racks in one zone and two racks in the other. We use multiple labels ("zone" in addition to "rack") in order to assign nodes to prefix groups.

```
{
    "networks": [
        {
            "name"      : "my-network",
            "cidr"      : "10.111.0.0/16",
            "block_mask": 29
        }
    ],
    "topologies": [
        {
            "networks": [ "my-network" ],
            "map": [
                {
                    "assignment": { "zone" : "zone-A" },
```

(continues on next page)

(continued from previous page)

```
                    "groups"    : [                              # addresses from 10.
↪111.0.0/17
                        {
                            "assignment": { "rack": "rack-3" },
                            "groups"    : []                     # addresses from 10.
↪111.0.0/19
                        },
                        {
                            "assignment": { "rack": "rack-4" },
                            "groups"    : []                     # addresses from 10.
↪111.32.0/19
                        },
                        {
                            "assignment": { "rack": "rack-7" },
                            "groups"    : []                     # addresses from 10.
↪111.64.0/19
                        },
                        {
                            "assignment": { "rack": "rack-9" },
                            "groups"    : []                     # addresses from 10.
↪111.96.0/19
                        }
                    ]
                },
                {
                    "assignment": { "zone" : "zone-B" },
                    "groups"    : [                              # addresses from 10.
↪111.128.0/17
                        {
                            "assignment": { "rack": "rack-17" },
                            "groups"    : []                     # addresses from 10.
↪111.128.0/18
                        },
                        {
                            "assignment": { "rack": "rack-22" },
                            "groups"    : []                     # addresses from 10.
↪111.192.0/18
                        }
                    ]
                }
            ]
        }
    ]
}
```

## 8.2 Route Publisher Add-on

For Kubernetes clusters installed in datacenters, it is useful to enable the Romana Route Publisher add-on. It is used to automatically announce routes for Romana addresses to your BGP- or OSPF-enabled router, removing the need to configure these manually.

Because the routes are for prefixes instead of precise /32 endpoint addresses, the rate and volume of routes to publish is reduced.

### 8.2.1 Configuration

The Romana Route Publisher uses BIRD to announce routes from the node to other network elements. Configuration is separated into two parts:

- a static `bird.conf` to describe the basic configuration of BIRD, ending with an `include`
- a dynamic `publisher.conf` that is used to generate a config containing routes for Romana addresses

When the pod first launches, BIRD is launched using the static configuration. Then, when new blocks of Romana addreses are allocated to a node, the dynamic configuration is generated with routes for those blocks, and BIRD is given a signal to reload its configuration.

If your configuration requires custom configuration per-node or per-subnet, there is a naming convention for the files that can be used to support this.

Both config files will look for a "best match" extension to the name first. When loading `x.conf` on a node with IP `192.168.20.30/24`, it will first look for:

- `x.conf.192.168.20.30` (IP suffix for node-specific config)
- `x.conf.192.168.20.0` (Network address suffix, for subnet-specific config)
- `x.conf`

### 8.2.2 Examples

**bird.conf (for both BGP and OSPF)**

```
router id from 192.168.0.0/16;

protocol kernel {
    scan time 60;
    import none;
        export all;
}

protocol device {
    scan time 60;
}

include "conf.d/*.conf";
```

- Make sure the CIDR specified for `router id` matches your cluster nodes.
- The `protocol kernel` and `protocol device` can be modified, or just deleted if not necessary.
- Add any additional, global BIRD configuration to this file (eg: debugging, timeouts, etc)
- The `include` line is the hook to load the generated dynamic config. It should be in your `bird.conf` exactly as specified.

**publisher.conf for OSPF**

```
protocol static romana_routes {
    {{range .Networks}}
    route {{.}} reject;
    {{end}}
}

```

(continues on next page)

(continued from previous page)

```
protocol ospf OSPF {
  export where proto = "romana_routes";
  area 0.0.0.0 {
    interface "eth0" {
      type broadcast;
    };
  };
}
```

- The first section, `protocol static static_bgp` is used by the `romana-route-publisher` to generate a dynamic config.
- The second section, `protocol ospf OSPF` should contain the `export` entry, and `area` blocks to match your environment.
- The interface names will need to be modified to match the node's actual interfaces
- Add any additional, protocol-specific BIRD configuration to this file

**publisher.conf for BGP**

```
protocol static romana_routes {
    {{range .Networks}}
    route {{.}} reject;
    {{end}}
}

protocol bgp BGP {
    export where proto = "romana_routes";
        direct;
    local as {{.LocalAS}};
    neighbor 192.168.20.1 as {{.LocalAS}};
}
```

- The first section, `protocol static static_bgp` is used by the `romana-route-publisher` to generate a dynamic config.
- The second section, `protocol bgp BGP` should be changed to match your specific BGP configuration.
- Add any additional, protocol-specific BIRD configuration to this file
- The `neighbor` address will likely be different for each subnet. To handle this, you can use multiple `publisher.conf` files with the appropriate network address suffixes, eg:
- bird.conf.192.168.20.0
- bird.conf.192.168.35.0

### 8.2.3 Installation

First, the configuration files need to be loaded into a `configmap`.

1. Put all the files into a single directory
2. `cd` to that directory
3. Run `kubectl -n kube-system create configmap route-publisher-config --from-file=.` (the `.` indicates the current directory)

Next, download the YAML file from here to your master node.

Then, load the Romana Route Publisher add-on by running this command on your master node.

```
kubectl apply -f romana-route-publisher.yaml
```

### 8.2.4 Verification

Check that route publisher pods are running correctly

```
$ kubectl -n kube-system get pods --selector=romana-app=route-publisher
NAME                            READY     STATUS    RESTARTS   AGE
romana-route-publisher-22rjh    2/2       Running   0          1d
romana-route-publisher-x5f9g    2/2       Running   0          1d
```

Check the logs of the bird container inside the pods

```
$ kubectl -n kube-system logs romana-route-publisher-22rjh bird
Launching BIRD
bird: Chosen router ID 192.168.XX.YY according to interface XXXX
bird: Started
```

Other messages you may see in this container:

```
bird: Reconfiguration requested by SIGHUP
bird: Reconfiguring
bird: Adding protocol romana_routes
bird: Adding protocol OSPF
bird: Reconfigured
```

Check the logs of the publisher container inside the pods

```
$ kubectl -n kube-system logs romana-route-publisher-22rjh publisher
Checking if etcd is running...ok.
member 8e9e05c52164694d is healthy: got healthy result from http://10.96.0.88:12379
cluster is healthy
Checking if romana daemon is running...ok.
Checking if romana networks are configured...ok. one network configured.
Checking for route publisher template....ok
Checking for pidfile from bird...ok
Launching Romana Route Publisher
```

Other messages you may see in this container:

```
20XX/YY/ZZ HH:MM:SS Starting bgp update at 65534 -> : with 2 networks
20XX/YY/ZZ HH:MM:SS Finished bgp update
```

These are normal, even if OSPF is being used.

## 8.3 Romana VIPs

Kubernetes users running on premises that want an easy way to expose their services outside a cluster on their datacenter network can use external-IPs.

Although external-IPs are simple, they represent a single point of failure for the service and require manual allocation and configuration on the nodes. When there are many to configure, this can be tedious and prone to error. Romana VIPs are a solution to these problems.

Romana VIPs are defined by an annotation in a service spec. Romana then automatically brings up that IP on a node. Romana chooses a node with a pod running locally to avoid network latency within the cluster. When a node with a Romana VIP fails, Romana will bring up the VIP on a new node, providing failover for external services.

Romana VIPs are useful for exposing services on datacenter LANs that only need simple kubeproxy load balancing across pods. Romana VIPs can also be used to expose individual pods when a stable IP is required, such as Cassandra and other Big Data applications. Romana VIPs work in conjunction with Romana DNS, which can be deployed as a service discovery mechanism for individual pods exposed outside of a cluster.

Romana VIP failover requires that all nodes be on the same network segment. Addresses for Romana VIPs must be manually provisioned on the network.

### 8.3.1 Example configuration

The example below shows a RomanaIP (192.168.99.101) configured on a node for the nginx service by adding the `romanaip` annotation to the spec.

```
...
kind: Service
metadata:
  name: nginx
  annotations:
    romanaip: '{"auto": false, "ip": "192.168.99.101"}'
...
```

The complete service spec is available here

## 8.4 Romana DNS

Romana DNS adds DNS support for Romana VIPs. It is drop in replacement for kube-dns.

### 8.4.1 Installation

**On Master node of kubernetes cluster**

- Make a note on number of replicas for kube-dns using following command:

  ```
  echo `kubectl get deploy -n kube-system kube-dns -o jsonpath="{.spec.replicas}"`
  ```

- Now set replicas for kube-dns to zero using following command:

  ```
  kubectl scale deploy -n kube-system kube-dns --replicas=0
  ```

- Wait till kube-dns replicas are zero (around a minute or so)

**On All nodes i.e master and compute nodes of the kubernetes cluster**

- Remove earlier docker images and replace it romana one using commands below:

```
docker rmi gcr.io/google_containers/k8s-dns-kube-dns-amd64:1.14.5
docker pull pani/romanadns
docker tag pani/romanadns:latest gcr.io/google_containers/k8s-dns-kube-dns-
↪amd64:1.14.5
```

- Now return back to master node for further commands

**On Master node of kubernetes cluster**

- Now assuming you had 2 replicas before, from first step above, we restore the replica count for kube-dns as follows:

```
kubectl scale deploy -n kube-system kube-dns --replicas=2
```

- Wait for a minute or so for the pod to come up and we have romanaDNS up and running.

### 8.4.2 DNS Testing

- Run dig to see if dns is working properly using command:

```
dig @10.96.0.10 +short romana.kube-system.svc.cluster.local
```

- Download this sample nginx yaml file and then use following command to create an nginx service with RomanaIP in it:

```
kubectl create -f nginx.yml
```

- This should create and load nginx service with RomanaIP, which should reflect in the dig result below:

```
dig @10.96.0.10 +short nginx.default.svc.cluster.local
```

### 8.4.3 Sample DNS Results

```
$ dig @10.96.0.10 +short romana.kube-system.svc.cluster.local
10.96.0.99
192.168.99.10
$ dig @10.96.0.10 +short nginx.default.svc.cluster.local
10.116.0.0
10.99.181.64
192.168.99.101
```

# CHAPTER 9

# Contact Us

- By email: info@romana.io
- On the Romana Slack. Please request an invite by email.
- On GitHub, just open an issue