(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
19 December 2024 (19.12.2024)

WIPO | PCT

(10) International Publication Number
WO 2024/254734 A1

(51) International Patent Classification:
*G06F 9/455* (2018.01) *H04L 41/0895* (2022.01)
*H04L 41/08* (2022.01)

(21) International Application Number:
PCT/CN2023/099675

(22) International Filing Date:
12 June 2023 (12.06.2023)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant *(for BH, KW only)*: **VMWARE INFORMATION TECHNOLOGY (CHINA) CO., LTD.** [CN/CN]; F/1, F/8, F/9, F/17, F/18, South Building, Tower C, Raycom InfoTech Park, No.2 Kexueyuan South Road, Haidian District, Beijing 100190 (CN).

(71) Applicant *(for all designated States except BE, BF, BH, BJ, CF, CG, CI, CM, CY, FR, GA, GN, GQ, GR, GW, IE, KM, KW, LT, LV, MC, ME, ML, MR, MT, NE, NL, SI, SM, SN, SZ, TD, TG)*: **VMWARE LLC** [US/US]; 3401 Hillview Avenue, Palo Alto, California 94304 (US).

(72) Inventors: **TIAN, Quan**; Level 8, S. Wing Tower C, Raycom Info Tech Park, No. 2 Kexueyuan S. Rd., Haidian District, Beijing 100190 (CN). **SHEN, Jianjun**; 3401 Hillview Avenue, Palo Alto, California 94304 (US). **DING, Yang**; 3401 Hillview Avenue, Palo Alto, California 94304 (US). **HAN, Donghai**; Level 8, S. Wing Tower C, Raycom Info Tech Park, No. 2 Kexueyuan S. Rd., Haidian District, Beijing 100190 (CN).

(74) Agent: **LEE AND LI - LEAVEN IPR AGENCY LTD.**; Unit 2202, Tower A, Beijing Pudi Hotel, No. 7 Jian Guo

(54) **Title:** LAYER 7 NETWORK SECURITY FOR CONTAINER WORKLOADS

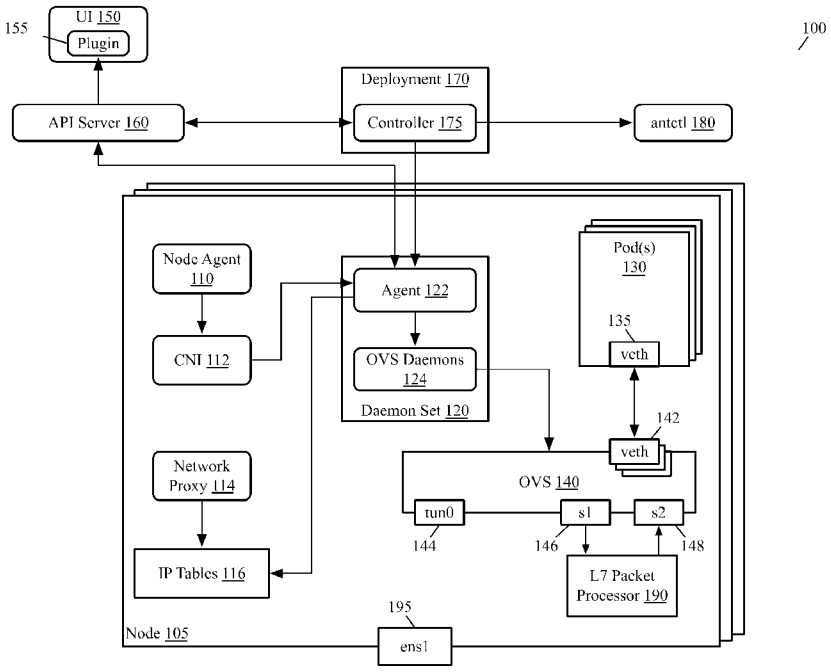


*Figure 1*

(57) **Abstract:** Some embodiments of the invention provide a method of performing layer 7 (L7) packet processing for a set of Pods executing on a host computer, the set of Pods managed by a container orchestration platform. The method is performed at the host computer. The method receives notification of a creation of a traffic control (TC) custom resource (CR) that is defined by reference to a TC custom resource definition (CRD). The method identifies a set of interfaces of a set of one or more managed forwarding elements (MFEs) executing on the host computer that are candidate interfaces for receiving flows that need to be directed based on the TC CR to a layer 7 packet processor. Based on the identified set of interfaces, the method provides a set of flow records to the set of MFEs to process in order to direct a subset of flows that the set of MFEs receive to the layer 7 packet processor.

WO 2024/254734 A1

*[Continued on next page]*

Men South Avenue, Dongcheng District, Beijing 100005 (CN).

**(81) Designated States** *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CV, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IQ, IR, IS, IT, JM, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, MG, MK, MN, MU, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

**(84) Designated States** *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, CV, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SC, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, ME, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

**Published:**

— *with international search report (Art. 21(3))*

# LAYER 7 NETWORK SECURITY FOR CONTAINER WORKLOADS

## BACKGROUD

**[0001]** In today's container networks (e.g., Kubernetes), network policies (e.g., Kubernetes Network Policies) are used to control traffic flow at Layer 3 (Network) and Layer 4 (Transport) in the OSI model, based on IP address, transport protocol, and port. Many cloud native applications use application layer protocols such as HTTP or gRPC to communicate and expose APIs. Securing such applications at the IP and port level provides very limited security as the APIs are either entirely exposed to a client or not exposed at all. Some solutions in the cloud native ecosystem provide the capability to filter on specific application protocol attributes, and achieve it by extending filters of the Envoy proxy, which introduces extra costs and cannot be extended to support more advanced network security features like protocol detection, Intrusion Detection (IDS) and Intrusion Prevention (IPS).

## BRIEF SUMMARY

**[0002]** Some embodiments of the invention provide a method of performing layer 7 (L7) packet processing for a set of Pods that execute on a host computer. The Pods are managed by a container orchestration platform in some embodiments. Also, in some embodiments, the method is performed by a module executing on the host computer. For instance, in some embodiments, the method is performed by a network plugin agent deployed to the host computer.

**[0003]** The method initially receives notification regarding the creation of a traffic control (TC) custom resource (CR) instance that is defined by reference to a TC custom resource definition (CRD). The traffic control CR uses one or more high-level attributes that define a category of data traffic flows that should be directed to a set of one or more L7 packet processor. The method receives the notification from the container orchestration platform in some embodiments. For example, the notification in some embodiments is received from an API server of the container orchestration platform. This API server receives an API (application programming interface) that directs the API server to create the TC CR, and creates the TC CR instance by parsing the API to extract parameters used to define the TC CR.

**[0004]** After receiving the notification regarding the creation of the TC CR, the method then uses the TC CR attributes to generate a set of flow records for a set of one or more managed

forwarding elements (MFEs) executing on the host computer to use to direct a set of flows received by the MFE set to the layer 7 packet processor. After generating the set of flow records, the method provides the set of flow records to the set of MFEs to use. For instance, in some embodiments, the set of MFEs includes an OVS (Open Virtual Switch) forwarding element executing on the host computer. In these embodiments, the method provides one or more OVS flow records for the OVS forwarding element to use to identify flows that need to be directed or re-directed to the L7 packet processor (e.g., an L7 firewall).

**[0005]** To generate the flow records, the method in some embodiments identifies a set of interfaces of the MFE set executing on the host computer that are candidate interfaces for receiving flows that need to be directed based on the TC CR instance to a layer 7 packet processor. The identified set of interfaces in some embodiments can include (1) an interface associated with a Pod that executes on the host computer and that is a source of a flow that needs to be directed to the L7 packet processor, and/or (2) an interface associated with a Pod that is a destination of a flow that needs to be directed to the L7 packet processor.

**[0006]** The TC CR instance includes one or more attributes that the method uses to identify source or destination Pods. These might be attributes that specifically identify these Pods, or might be high-level attributes that identify the Pods more generally. For instance, these attributes in some embodiments include labels associated with Pods, or labels associated with namespaces. These attributes can also specify a direction of flows of interest (e.g., in, out, or both). In some embodiments, whether a Pod is a source or destination of a flow is determined based on whether the TC CR instance is specified for ingress flows to the Pod (i.e., direction specified is "in"), egress flows from the Pod (i.e., direction specified is "out"), or both ingress and egress flows associated with the Pod (i.e., direction specified is "both"). The source or destination Pods, in some embodiments, are defined in the TC CR instance using high level attributes, while the set of flow records specifies interfaces of the source or destination Pods in terms of network addresses associated with the source or destination Pods (i.e., lower-level attributes).

**[0007]** In addition to identifying the set of interfaces connected to the source or destination machines (e.g., Pods), the method in some embodiments also identifies the MFE interface associated with the L7 packet processor. For instance, in some embodiments, the TC CR instance has a set of one or more identifier attributes (e.g., name or label attribute) that the method uses to identify the L7 packet processor to use, and the method uses the identifier attribute set to identify

an MFE interface associated with the L7 packet processor. The method in some embodiments uses the identified MFE interface associated with the L7 packet processor to define the flow record(s) that it generates for directing or re-directing a set of flows to the L7 packet processor. For instance, some embodiments use an IP (Internet protocol) address assigned to the identified MFE interface for the L7 packet processor as the IP address to which packets matching a flow record in the set of flow records are to be directed. In some embodiments, the L7 packet processor executes on the host computer, while in other embodiments, the L7 packet processor executes outside of the host computer.

**[0008]** In some embodiments, the L7 packet processor performs one or more middlebox service operations based on L7 parameters extracted from flows directed to and processed by the L7 packet processor. The L7 parameters are extracted by the L7 packet processor of some embodiments during a deep packet inspection (DPI) operation performed on the flows by the L7 packet processor. Examples of middlebox service operations include firewall operations, load balancing operations, network address translation operations, intrusion detection operations, and intrusion prevention operations, in some embodiments.

**[0009]** In some embodiments, the flow record that the method generates for and provides to the MFE has an action attribute that specifies how the MFE should forward a flow to the L7 packet processor. The method generates this action attribute based on an action attribute that is specified in the TC CR instance in some embodiments. Examples of such actions include (1) "redirect" which directs the MFE to forward a particular flow to the L7 packet processor, and (2) "copy" or "mirror" which directs the MFE to forward a copy of a flow to the L7 packet processor. When a flow's packet is redirected to the L7 packet processor, the MFE would receive the packet again from the L7 packet processor (assuming that the packet is not dropped), and then forward the packet to the destination specified by the packet's source (e.g., the source Pod).

**[0010]** The preceding Summary is intended to serve as a brief introduction to some embodiments of the invention. It is not meant to be an introduction or overview of all inventive subject matter disclosed in this document. The Detailed Description that follows and the Drawings that are referred to in the Detailed Description will further describe the embodiments described in the Summary as well as other embodiments. Accordingly, to understand all the embodiments described by this document, a full review of the Summary, the Detailed Description, the

Drawings, and the Claims is needed. Moreover, the claimed subject matters are not to be limited by the illustrative details in the Summary, the Detailed Description, and the Drawings.

## BRIEF DESCRIPTION OF FIGURES

**[0011]** The novel features of the invention are set forth in the appended claims. However, for purposes of explanation, several embodiments of the invention are set forth in the following figures.

**[0012]** **Figure 1** illustrates a networking solution for a container orchestration network of some embodiments.

**[0013]** **Figure 2** illustrates an example of a control system of some embodiments of the invention that processes APIs for a container orchestration network.

**[0014]** **Figure 3** illustrates an example of the structure of a traffic control CRD of some embodiments.

**[0015]** **Figure 4** illustrates an example of a CR generated based on a CRD of some embodiments.

**[0016]** **Figure 5** illustrates an example of calculated OpenFlow rules of some embodiments for first and second Pods having respective OVS ports.

**[0017]** **Figure 6** conceptually illustrates a diagram showing traffic flows between Pods executing on a host that includes an L7 packet processing engine.

**[0018]** **Figure 7** conceptually illustrates a diagram showing a traffic flow of some embodiments between Pods on different hosts that both include an L7 packet processing engine.

**[0019]** **Figure 8** conceptually illustrates a process performed in some embodiments for performing traffic control using a traffic control CR.

**[0020]** **Figure 9** conceptually illustrates a computer system with which some embodiments of the invention are implemented.

## DETAILED DESCRIPTION

**[0021]** In the following detailed description of the invention, numerous details, examples, and embodiments of the invention are set forth and described. However, it will be clear and apparent to one skilled in the art that the invention is not limited to the embodiments set forth and that the invention may be practiced without some of the specific details and examples discussed.

**[0022]** Some embodiments of the invention provide a method of performing layer 7 (L7) packet processing for a set of Pods that execute on a host computer. The Pods are managed by a container orchestration platform in some embodiments. Container orchestration platforms of some embodiments are central platforms for managing the lifecycle of containers, as well as providing a network so that the containers can communicate. In some embodiments, container orchestration platforms provide an automated solution for running the containerized workloads and services, such as provisioning, deploying, scaling (e.g., scaling up or down), networking, and load balancing. Container orchestration platforms allow for simplification of large amounts of complex operations introduced by containers, resilience in the form of container orchestration tools that automatically restart or scale a container or cluster, and additional security by reducing or eliminating human error through automation.

**[0023]** Containers are lightweight, ephemeral software packages that include everything required to run an application (e.g., application code and everything needed for the code to run properly), while being decoupled from the underlying operating system (OS) and infrastructure on which they run. As a result, containers are highly portable, speed up application development and deployment, and consume fewer resources. In some embodiments, one or more containers are grouped into Pods, in which storage and network resources, as well as a specification for how to run the containers, are shared. The contents of a Pod are co-located, co-scheduled, and run in a shared context.

**[0024]** Kubernetes is an example of an open-source container orchestration platform utilized in some embodiments. Microservices of some embodiments can be modelled as Kubernetes Services over deployments of Pods, which specifies how to expose and scale a fleet of containers created for the services of some embodiments. However, because Kubernetes does not provide a default implementation for networking among its workloads, the networking implementation is delegated to a CNI (Container Network Interface) plugin. The CNI plugin implements standard CNI API (application programming interface) set forth by Kubernetes, and configures networking for each Pod created in a Kubernetes cluster (e.g., a set of nodes that run containerized applications). In some embodiments, CNIs can also choose to implement Kubernetes NetworkPolicy, which is a declarative way of specifying Layer 3 (Network) and Layer 4 (Transport) networking security posture of Pod workloads, and thus the accessibility of the service that exposes these Pods.

**[0025]** An example of a CNI plugin used in some embodiments for network implementation is Antrea. Antrea is a performant CNI plugin that includes many features, leverages Open vSwitch (OVS) as the data plane, and efficiently implements Kubernetes NetworkPolicy using conjunctive OpenFlow (i.e., a communications protocol that provides access to the forwarding plane of a network switch or router over a network). In addition to the standard Kubernetes NetworkPolicy, which focuses on expressing a developer's intent to secure their applications, Antrea also offers an extended set of native policy constructs that fit the cluster administrator persona. Antrea-native policies support prioritization of rules, grouping policies with tiers, audit logging, and FQDN-based (fully-qualified-domain-name-based) filtering. Similar to Kubernetes NetworkPolicy, this extended set of administrator policies also operates on Layer 3 (L3) and Layer 4 (L4) only, and as such, networking traffic is either allowed or entirely blocked for a specific host port and protocol combination of a service backend.

**[0026]** In some embodiments, the method is performed by a module executing on the host computer. For instance, the method of some embodiments is performed by a network plugin agent deployed to the host computer. The network plugin agent, in some embodiments, is a network plugin that interfaces with the CNI delegated to implement network services for the container orchestration platform. In some embodiments, the network plugin can define forwarding rules for MFEs (managed forwarding elements), such as OVS, executing on host computers along with containers managed by the container orchestration platform (e.g., Kubernetes).

**[0027]** The network plugin of some embodiments can also define L3/L4 services (e.g., firewall services, load balancing services, etc.). Moreover, in some Kubernetes deployments, Kubernetes NetworkPolicies can be used to define L3/L4 firewall rules, as mentioned above. However, all of these rules are L3/L4 services, and do not include L7 services (e.g., application layer services). Some embodiments provide a novel method for allowing a network administrator to use a traffic control (TC) custom resource (CR) to efficiently specify re-directing and/or mirroring policies in terms of high-level constructs, and having the network plugin convert these TC CR instance constructs into redirecting policies and/or mirroring policies that redirect or mirror flows to the L7 packet processor or processors that perform L7 services.

**[0028]** The method initially receives notification regarding the creation of a TC CR instance that is defined by reference to a TC custom resource definition (CRD). The traffic control CR uses

one or more high-level attributes that define a category of data traffic flows that should be directed to a set of one or more L7 packet processor. The method receives the notification from the container orchestration platform in some embodiments. For example, the notification in some embodiments is received from an API server of the container orchestration platform. This API server receives an API (application programming interface) that directs the API server to create the TC CR, and creates the TC CR instance by parsing the API to extract parameters used to define the TC CR.

**[0029]** After receiving the notification regarding the creation of the TC CR, the method then uses the TC CR attributes to generate a set of flow records for a set of one or more managed forwarding elements (MFEs) executing on the host computer to use to direct a set of flows received by the MFE set to the layer 7 packet processor. After generating the set of flow records, the method provides the set of flow records to the set of MFEs to use. For instance, in some embodiments, the set of MFEs includes an OVS (Open Virtual Switch) forwarding element executing on the host computer. In these embodiments, the method provides one or more OVS flow records for the OVS forwarding element to use to identify flows that need to be directed or re-directed to the L7 packet processor (e.g., an L7 firewall).

**[0030]** To generate the flow records, the method identifies a set of interfaces of the MFE set executing on the host computer that are candidate interfaces for receiving flows that need to be directed based on the TC CR instance to a layer 7 packet processor. The identified set of interfaces in some embodiments can include (1) an interface associated with a Pod that executes on the host computer and that is a source of a flow that needs to be directed to the L7 packet processor, and/or (2) an interface associated with a Pod that is a destination of a flow that needs to be directed to the L7 packet processor.

**[0031]** The TC CR instance includes one or more attributes that the method uses to identify source or destination Pods. These might be attributes that specifically identify these Pods, or might be high-level attributes that identify the Pods more generally. For instance, these attributes in some embodiments include labels associated with Pods, or labels associated with namespaces. These attributes can also specify a direction of flows of interest (e.g., in, out, or both). In some embodiments, whether a Pod is a source or destination of a flow is determined based on whether the TC CR instance is specified for ingress flows to the Pod (i.e., direction specified is “in”), egress flows from the Pod (i.e., direction specified is “out”), or both ingress and egress flows

associated with the Pod (i.e., direction specified is "both"). The source or destination Pods, in some embodiments, are defined in the TC CR instance using high level attributes, while the set of flow records specifies interfaces of the source or destination Pods in terms of network addresses associated with the source or destination Pods (i.e., lower-level attributes).

**[0032]** In addition to identifying the set of interfaces connected to the source or destination machines (e.g., Pods), the method of some embodiments also identifies the MFE interface associated with the L7 packet processor. For instance, in some embodiments, the TC CR instance has a set of one or more identifier attributes (e.g., name or label attribute) that the method uses to identify the L7 packet processor to use, and the method uses the identifier attribute set to identify an MFE interface associated with the L7 packet processor. The method in some embodiments uses the identified MFE interface associated with the L7 packet processor to define the flow record(s) that it generates for directing or re-directing a set of flows to the L7 packet processor. For instance, some embodiments use an IP (Internet protocol) address assigned to the identified MFE interface for the L7 packet processor as the IP address to which packets matching a flow record in the set of flow records are to be directed. In some embodiments, the L7 packet processor executes on the host computer, while in other embodiments, the L7 packet processor executes outside of the host computer.

**[0033]** Since cloud native applications typically use application protocols such as HTTP (hypertext transfer protocol) or gRPC (Google remote procedure call) to expose their APIs, some embodiments provide means for developers to deploy application-aware Layer 7 (Application Layer or L7) policies that filter requests to their services based on specific application protocol attributes (e.g., HTTP verbs). In some embodiments, a traffic control and processing model for L7 security services is provided with the CNI plugin (e.g., Antrea) by introducing a new API that allows controlling the traffic of a container orchestration network's workloads and leveraging L7 packet processing engines for deep packet inspection (DPI) and policy enforcement.

**[0034]** For example, in some embodiments, the L7 packet processor performs one or more middlebox service operations based on L7 parameters extracted from flows directed to and processed by the L7 packet processor. The L7 parameters are extracted by the L7 packet processor of some embodiments during a DPI operation performed on the flows by the L7 packet processor. Examples of middlebox service operations include firewall operations, load balancing

operations, network address translation operations, intrusion detection operations, and intrusion prevention operations, in some embodiments.

**[0035]** In some embodiments, the flow record that the method generates for and provides to the MFE has an action attribute that specifies how the MFE should forward a flow to the L7 packet processor. The method generates this action attribute based on an action attribute that is specified in the TC CR instance in some embodiments. Examples of such actions include (1) "redirect" which directs the MFE to forward a particular flow to the L7 packet processor, and (2) "copy" or "mirror" which directs the MFE to forward a copy of a flow to the L7 packet processor. When a flow's packet is redirected to the L7 packet processor, the MFE would receive the packet again from the L7 packet processor (assuming that the packet is not dropped), and then forward the packet to the destination specified by the packet's source (e.g., the source Pod).

**[0036]** **Figure 1** illustrates a networking solution of some embodiments. The networking solution, in some embodiments, is an Antrea networking solution implemented for a container orchestration network (e.g., Kubernetes). In some embodiments, as a Kubernetes networking solution, Antrea implements the CNI, while Kubernetes NetworkPolicy operates at L3/L4 to provide network connectivity and security services for a Kubernetes cluster (i.e., collection of nodes for running containerized applications), leveraging the benefit of programmable networks from OVS to Kubernetes. OVS is a widely adopted high-performance programmable virtual switch, originating from VMware, Inc., that is designed to enable effective network automation through programmatic extensions. The network solution described herein leverages OVS in its architecture to efficiently implement Pod networking and security features.

**[0037]** In some embodiments, because of the programmable OVS, forwarding functions are opened to programmatic extension and control. Based on this, a flexible IPAM (IP address management) plugin (e.g., Antrea IPAM plugin) overrides and extends the existing flow tables in some embodiments, which are managed by a centralized CRD (e.g., instead of a local store IP management state from the original host-local IPAM plugin). This centralized controller helps to provide the ability of multiple networks on Pod and IPAM per-namespace, according to some embodiments. In some embodiments, in an L3 forwarding table, all traffic destined to a remote Pod is forwarded through the appropriate tunnel, and for the return flow from a remote Pod to a local node, a distinction is drawn between the remote gateway and the local gateway.

**[0038]** As shown, the networking solution 100 includes nodes 105 (e.g., Kubernetes nodes), a user interface (UI) 150 with a plugin 155 (e.g., Antrea plugin), an API server 160 (e.g., Kubernetes API server), a deployment 170 that runs the controller 175 (e.g., Antrea controller), and command-line tool (antctl) 180 (e.g., Antrea command-line tool). In some embodiments, the UI 150, API server 160, deployment 170, and a command-line tool 180 execute together as part of the control plane on a single master node.

**[0039]** To provide a more flexible IPAM (host-local IP address management) that is based on namespace isolation, the deployment 170 of some embodiments runs the controller 175. The controller 175 is used along with corresponding CRDs to manage all of the IP addresses for Pods executing on nodes in the network. As a result, each Pod subnet is associated with a respective namespace such that the IP assigned to a Pod is related to its business, in some embodiments.

**[0040]** Additionally, Pods located under the same namespace are in the same local area network (LAN) in some embodiments, while Pods under different namespaces are isolated on different networks. In some embodiments, a static IP address assigned to a Pod is configured by the annotation filed for the corresponding configuration file. Users (e.g., administrators) of some embodiments are also able to monitor the IP usage from the command-line tool 180 or the UI 150 in order to expand the corresponding IP resource pool in a timely manner when IP resources are exhausted.

**[0041]** In some embodiments, for each container orchestration network cluster that runs Antrea as CNI, Antrea creates a single deployment of an Antrea controller image (e.g., controller 175 of the deployment 170), as well as a daemon set 120 that includes an agent 122 (e.g., an Antrea network plugin agent) and OVS daemon containers 124 on every node 105 of the cluster. The controller 175 serves as the central control plane that listens to resource updates (e.g., Network Policy updates and Antrea-native Policy updates) from the API server 160, computes the desired state for the OVS datapath of each node 105, and distributes the information to each agent 122 that manages workloads affected by the change.

**[0042]** The agent 122 of some embodiments is responsible for managing Pod network interfaces and OVS flows. The agent 122 creates the OVS bridge 140, a veth pair for each Pod (e.g., a veth 135 of a Pod 130 and a veth 142 of the OVS bridge 140), with one end being in the Pod's network namespace and the other connected to the OVS bridge. On the OVS bridge 140, the agent 122 of some embodiments also creates a tunnel port 144 (e.g., antrea-tun0) for creating

overlay tunnels to other nodes, and, while not shown, creates an internal port (e.g., antrea-gw0 by default) to be the gateway of the node's subnet. The agent 122 of some embodiments registers for notifications regarding TrafficControl CRs from the API server 160 and translates high-level definitions of attributes from the TrafficControl CRs to lower-level network interface attributes (e.g., IP addresses of source and destination interfaces, L7 packet processor interfaces, etc.), and creates OVS flows (e.g., OpenFlow rules) for redirecting or mirroring certain flows to the L7 packet processor.

**[0043]** The UI 150 is used to manage Kubernetes clusters by translating human-readable commands into API calls that can be understood by the Kubernetes API server 160. In some embodiments, the UI 150 is a VMware Octant UI, and presents its output in a graphical user interface (GUI) for viewing by a user (e.g., administrator). The UI 150 runs locally on the user's workstation, according to some embodiments, and as a result, does not use up resources of the node or nodes that it manages. The UI 150 includes plugin 155 (e.g., Antrea plugin) for receiving CRDs (e.g., Antrea CRDs) from the API server 160.

**[0044]** The controller 175 additionally monitors network policy, pod, and namespace resources with the API server 160. The controller 175, in some embodiments, uses information associated with these resources to compute policy rules, which can be translated to OVS flows efficiently and disseminated to a targeted agent (e.g., agent 122) that runs on a node along with one or more affected Pods. The API server 160 enables different components of the cluster (i.e., a master node and set of one or more worker nodes) to communicate with each other and with components external to the cluster, according to some embodiments. Additionally, in some embodiments, the API server 160 enables users to query and alter the states of API objects, such as Pods, namespaces, configuration maps, and events.

**[0045]** Each of the worker nodes 105 includes a node agent 110 (e.g., a Kubernetes kubelet), CNI 112 (e.g., an Antrea CNI), network proxy 114 (e.g., a Kubernetes kube-proxy), IP tables 116, daemonset 120, one or more Pods 130, and an OVS bridge 140. The node agent 110, in some embodiments, is responsible for registering the node 105 with the API server 160. Additionally, the node agent 110 ensures that containers defined in Pod specifications received from the API server 160 are both running and healthy. In some embodiments, instead of receiving the Pod specifications from the API server 160, the node agent 110 receives the Pod specifications from an HTTP endpoint (not shown) or an HTTP server (not shown).

**[0046]** The daemonset 120 includes two containers to run the agent 122 and the OVS daemons 124, respectively, on every node, as well as an init-container (not shown) that installs the CNI 112 on the node. The CNI 112, in some embodiments, requests IP addresses for Pods instantiated on the node 105, and interacts with the agent 122 to update the IP tables 116 with the assigned IP addresses.

**[0047]** The network proxy 114 runs on the node 105 to maintain network rules on the node to allow network communications to the Pods 130 from sessions within the cluster, as well as sessions outside of the cluster. In some embodiments, the network proxy 114 forwards data traffic for the Pods itself using the IP addresses in the IP tables 116. OVS of some embodiments realizes the data plane on each of the worker nodes 105 at the same time, and in response, the controller 175 implements the control plane of the software-defined network (SDN) for which the networking solution 100 is implemented.

**[0048]** The agent 122 helps to bridge the controller 175 and OVS between the master node (not shown) and each other node 105 by creating the OVS bridge 140 and a veth pair for each Pod 130, with one end 135 of the veth pair being in the pod's network namespace, and the other end 142 connected to the OVS bridge 140. As shown, the agent 122 interacts with the OVS bridge 140 via the OVS daemons 124. In some embodiments, on the OVS bridge 140, the agent 122 also creates an internal port (not shown) by default as the gateway of the node's subnet, and a tunnel port (not shown) for creating overlay tunnels to other nodes 105. For instance, in the Antrea networking solution, "antrea-gw0" is created as an internal port by default as the gateway of the node's subnet, and "antrea-tun0" is created as a tunnel port for creating the overlay tunnels to other nodes. Additional details regarding per-namespace IP address management can be found in U.S. Patent Application 17/684,160, titled "Per-Namespace IP Address Management Method for Container Networks", and filed March 1, 2022. U.S. Patent Application 17/684,160 is incorporated herein by reference.

**[0049]** **Figure 2** illustrates an example of a control system 200 of some embodiments of the invention that processes APIs for a container orchestration network. The control system 200 of some embodiments specifically processes APIs that use the Kubernetes-based declarative model to describe the desired state of (1) the endpoints (e.g., machines) to deploy, and (2) the connectivity, security and service operations that are to be performed for the deployed endpoints (e.g., private and public IP addresses connectivity, load balancing, security policies, etc.).

**[0050]** To process the APIs, the control system 200 uses one or more CRDs to define some of the resources referenced in the APIs. The system 200 performs automated processes to deploy a logical network that connects the deployed endpoints (e.g., machines) and segregates these endpoints from other endpoints in the datacenter set. The endpoints are connected to the deployed logical network of a virtual private cloud (VPC) in some embodiments. In some embodiments, the control system 200 instead processes APIs that reference the CRDs.

**[0051]** As shown, the control system 200 includes two or more master nodes 235 for API processing, an SDN manager cluster 210, and an SDN controller cluster 215. Each of the master nodes 235 for API processing includes an API processing server 240, a node agent 242, compute managers and controllers 217, and a container plugin 245. In some embodiments, the container plugin 245 is a network controller plugin (NCP). The API processing server 240 receives intent-based API calls and parses these calls. The received API calls in some embodiments are in a declarative, hierarchical Kubernetes format, and may contain multiple different requests.

**[0052]** The API processing server 240 parses each received intent-based API request into one or more individual requests. When the requests relate to the deployment of endpoints, the API server provides these requests directly to compute managers and controllers 217, or indirectly provide these requests to the compute managers and controllers 217 through the node agent 242 and/or the container plugin 245 running on the master node 235 (e.g., a Kubernetes master node). The compute managers and controllers 217 then deploy VMs (virtual machines) and/or sets of containers on host computers in the availability zone.

**[0053]** The node agent 242 on a node can register the node with the API server 240 using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider. The node agent 242 receives sets of containerspecs, YAML (a data serialization language) or JavaScript Object Notation (JSON) formatted objects that each describe a pod. The node agent 242 uses sets of containerspecs to create (e.g., using the compute managers and controllers 217) the sets of containers that are provided by various mechanism elements (e.g., from the API server 240) and ensures that the containers described in those sets of containerspecs are running and healthy.

**[0054]** The API calls can also include requests that require network elements to be deployed. In some embodiments, these requests explicitly identify the network elements to deploy, while in other embodiments the requests can also implicitly identify these network elements by requesting the deployment of compute constructs (e.g., compute clusters, containers, etc.) for which

network elements have to be defined by default. The control system 200 of some embodiments uses the container plugin 245 to identify the network elements that need to be deployed, and to direct the deployment of these network elements.

**[0055]** In some embodiments, the API calls refer to extended resources that are not defined per se by the baseline Kubernetes system. For these references, the API server 240 uses one or more CRDs 220 to interpret the references in the API calls to the extended resources. The CRDs in some embodiments define extensions to the Kubernetes networking requirements (e.g., new resources in a Kubernetes environment). In some embodiments, the CRDs can include network-attachment-definition (NDs), Virtual Network Interfaces (VIF) CRDs, Virtual Network CRDs, Endpoint Group CRDs, security CRDs, Virtual Service Object (VSO) CRDs, and Load Balancer CRDs. As described herein, the CRDs of some embodiments define traffic control. In some embodiments, the CRDs are provided to the API processing server 240 in one stream with the API calls.

**[0056]** The container plugin 245 is the interface between the API server 240 and the SDN manager cluster 210 that manages the network elements that serve as the forwarding elements (e.g., switches, routers, bridges, etc.) and service elements (e.g., firewalls, load balancers, etc.) in an availability zone. The SDN manager cluster 210 directs the SDN controller cluster 215 to configure the network elements to implement the desired forwarding elements and/or service elements (e.g., logical forwarding elements and logical service elements) of one or more logical networks. The SDN controller cluster 215 interacts with local controllers on host computers and edge gateways to configure the network elements in some embodiments.

**[0057]** In some embodiments, the container plugin 245 registers for event notifications with the API server 240, e.g., sets up a long-pull session with the API server to receive all CRUD (Create, Read, Update and Delete) events for various CRDs that are defined for networking. In some embodiments, the API server 240 is a Kubernetes master VM, and the container plugin 245 runs in this VM as a Pod. The container plugin 245 in some embodiments collects realization data from the SDN resources for the CRDs and provides this realization data as it relates to the CRD status. In some embodiments, the container plugin 245 communicates directly with the API server 240 and/or through the node agent 242.

**[0058]** The container plugin 245 in some embodiments processes the parsed API requests relating to NDs, VIFs, virtual networks, load balancers, endpoint groups, security policies, and

VSOs, to direct the SDN manager cluster 210 to implement (1) the NDs that designate network segments for use with secondary interfaces of sets of containers, (2) the VIFs needed to connect VMs and sets of containers to forwarding elements on host computers, (3) virtual networks to implement different segments of a logical network of the VPC, (4) load balancers to distribute the traffic load to endpoints, (5) firewalls to implement security and admin policies, and (6) exposed ports to access services provided by a set of endpoints in the VPC to endpoints outside and inside of the VPC. In some embodiments, rather than directing the SDN manager cluster 210 to implement the NDs, VIFs, virtual networks, load balancers, endpoint groups, security policies, and VSOs, the container plugin 245 communicates directly with the SDN controller cluster 215 to direct the SDN controller cluster 215 to implement the NDs, VIFs, virtual networks, load balancers, endpoint groups, security policies, and VSOs.

**[0059]** The API server 240 provides the CRDs 220 that have been defined for these extended network constructs to the container plugin 245 for it to process the APIs that refer to the corresponding network constructs (e.g., network segments). The API server 240 also provides configuration data from the configuration storage 225 to the container plugin 245. The configuration data in some embodiments include parameters that adjust the pre-defined template rules that the container plugin 245 follows to perform its automated processes. In some embodiments, the configuration data includes a configuration map. The configuration map of some embodiments may be generated from one or more directories, files, or literal values. In some embodiments, the configuration map is generated from files in the configuration storage 225, from data received by the API server from the container plugin and/or from data generated by the SDN manager 210. The configuration map in some embodiments includes identifiers of pre-created network segments of the logical network.

**[0060]** The container plugin 245 performs these automated processes to execute the received API requests in order to direct the SDN manager cluster 210 to deploy the network elements for the VPC. For a received API, the control system 200 performs one or more automated processes to identify and deploy one or more network elements that are used to implement the logical network for a VPC. The control system performs these automated processes without an administrator performing any action to direct the identification and deployment of the network elements after an API request is received.

**[0061]** The SDN managers 210 and controllers 215 can be any SDN managers and controllers available today. In some embodiments, these managers and controllers are the NSX-T managers and controllers licensed by VMware, Inc. In such embodiments, the container plugin 245 detects network events by processing the data supplied by its corresponding API server 240, and uses NSX-T APIs to direct the NSX-T manager 210 to deploy and/or modify NSX-T network constructs needed to implement the network state expressed by the API calls. The communication between the container plugin 245 and NSX-T manager 210 is asynchronous, in which the container plugin provides the desired state to NSX-T managers, which then relay the desired state to the NSX-T controllers to compute and disseminate the state asynchronously to the host computer, forwarding elements and service nodes in the availability zone (i.e., to the SDDC (software-defined datacenter) set controlled by the controllers 215).

**[0062]** After receiving the APIs from the container plugin 245, the SDN managers 210 in some embodiments direct the SDN controllers 215 to configure the network elements to implement the network state expressed by the API calls. In some embodiments, the SDN controllers serve as the central control plane (CCP) of the control system 200. An Antrea-NSX-T adapter running on a master node 235, in some embodiments, receives the parsed API requests and other data from the API server 240 (i.e., as opposed to the container plugin receiving the parsed API requests and other data), such as API requests for adding routable subnets for logical networks, and generates API calls to direct the manager and controller clusters 210 and 215 to implement the routable segments.

**[0063]** In some embodiments, by default, intra-node Pod traffic is transmitted from a source Pod to a destination Pod directly via the OVS bridge, while inter-node Pod traffic is transmitted through a tunnel between the two OVS bridges on two nodes. L3 and L4 network policies are enforced when the packets hit corresponding OpenFlow entries installed on OVS tables as they enter or leave the bridge, in some embodiments. To implement L7 security policies, in some embodiments, CNIs use proxying for cluster network traffic, where packets are intercepted and examined to verify if any L7 policy applies. Envoy, for example, is a proxy designed for large microservice mesh architectures in some embodiments.

**[0064]** Certain network virtualization and security platforms (e.g., NSX-T from VMware, Inc.) support distributed L7 firewalls and distributed IDS/IPS. For both features, these network virtualization and security platforms run proprietary L7 engines in the user space that perform

DPI and enforce L7 policies on the traffic, according to some embodiments. For example, on both ESX and KVM, NSX-T implements a separate channel to redirect traffic from the kernel data path (e.g., ESX vSwitch in ESX vmkernel, or OVS kernel data path for KVM) to the user space L7 engines. On KVM, the traffic redirection leverages the OVS Openflow packet-in mechanism, which can send matched packets from OVS to the Openflow controller through a Unix Domain socket. However, this approach is not performant, and it requires changes to L7 engines to receive the redirected packets. The NSX ESX implementation builds a proprietary packet redirection channel but has similar limitations.

**[0065]** In some embodiments, Antrea is extended to work with Envoy by redirecting inter-workload traffic towards the proxy. Alternatively, other embodiments take advantage of an L7 packet processing engine that does DPI and takes necessary actions for L7 rule hits, without terminating and re-establishing the connections or interfering with unrelated traffic. An example of an open-source network threat detection engine for L7 packet processing and DPI is Suricata. Suricata is a signature-based security platform that has automatic application layer protocol detection. Controllers can program rulesets by binding actions with L7 attribute matching criteria. Suricata also has a performant multi-threaded nature and comprehensive support for application layer protocols.

**[0066]** Suricata AF_PACKET capture is an efficient way to direct packets to the engine, and it supports the in-line mode mentioned above. This packet capture pattern requires a pair of dedicated interfaces so that any packet received on one interface is automatically sent to another. Two OVS internal ports are created on each node and attached to the same OVS bridge that the Pods are connected to, in some embodiments. By assigning these interfaces to Suricata via a configuration file, Suricata captures traffic from either interface, in some embodiments, and copies the packets from one to the other if they are not dropped by any rule.

**[0067]** Because, by default, Pod traffic is transmitted from source to destination directly, in order to enforce an L7 firewall for specific Pods, a TrafficControl CR that selects the Pods and redirects ingress traffic to one of the above network interfaces to which Suricata is listening is utilized in some embodiments. In addition, some embodiments install extra OpenFlow rules for the traffic that are sent back to the OVS bridge by Suricata after filtering, to prevent the traffic from being redirected again, leading to a loop.

**[0068]** In some embodiments, Suricata uses signatures to match known threats, policy violations, and malicious behavior. A signature consists of three parts, according to some embodiments. The first part is the action, which determines what happens when the signature matches. The second part is the header, which defines the protocol, IP addresses, ports, and direction of the rule. The third part is the rule options, which define the specifics of the rule. In some embodiments, signatures are used to instruct Suricata how to process the traffic. An example of HTTP specific signature is as follows:

drop http $EXTERNAL_NET any -> $POD_IP any (content:"/admin"; http_uri; content:"POST"; http_method; sid:1;)

**[0069]** With the above configuration, some embodiments ensure that POST requests from external networks to the URL "/admin" of the specific Pod are not allowed while it is open to an internal network, regardless of the port used. Furthermore, as Pods are considered ephemeral, and users may customize their own security policies via a Layer 7 NetworkPolicy API, in some embodiments, signatures are expected to be updated frequently along with the events of Pods and security policies. A signature synchronizer is added, in some embodiments, for translating the user-configured Layer 7 policies and the Pods they select to IP-based signatures, and reloading Suricata to take effect. In some embodiments, Suricata is used as the L7 packet processing engine and is integrated with Antrea to implement an L7 firewall, as will be further described below.

**[0070]** To provide application layer network security services to Kubernetes workloads with Antrea, some embodiments redirect network traffic entering or leaving containers to an application-aware packet processing engine for security inspection and traffic filtering. In some embodiments, this is used as the foundation to support L7 firewalls and network-based IPS. As the L7 packet processing engine is placed in-line between workloads, or between workloads and an external network, it can proactively block or deny malicious traffic in some embodiments based on security policies. For IDS and network security monitoring (NSM) where no network traffic alternation is required, some embodiments mirror the network traffic to the L7 packet processing engine for out-of-band security inspection.

**[0071]** In order to achieve this workflow, in some embodiments, there are two main modules, traffic control API layer and L7 packet processing engine. The traffic control API layer is responsible for steering specific traffic to the L7 packet processing engine, and the L7 packet

processing engine takes care of inspecting the traffic, detecting and stopping potential incidents, according to some embodiments.

**[0072]** In some embodiments, a traffic control API is added using Kubernetes CRD. When an API server receives a traffic control API, the API server uses a traffic control CRD that defines a CR and that is referenced by the traffic control API to create a traffic control CR instance based on the resource defined by the traffic control CRD, according to some embodiments. The traffic control API of some embodiments accepts client requests and controls the container traffic with OpenFlow rules. In some embodiments, the API is generic and thus provides a mechanism to specify the Pods whose traffic should be selected, the direction of the traffic, whether the traffic should be mirrored or redirected, and the network device port to which the packets should be mirrored or redirected.

**[0073]** **Figure 3** illustrates an example of the structure of a traffic control CRD 300 of some embodiments. As shown, the structure of the traffic control CRD 300 includes a specification for the desired behavior of traffic control, a traffic control specification structure, a namespace selector, a Pod selector, the direction of traffic that should be matched (i.e., in, out, or both), the action that should be taken for the matched traffic, and the device to which the matched traffic should be redirected or mirrored.

**[0074]** **Figure 4** illustrates an example of a CR 400 generated based on a CRD of some embodiments. As shown, the CR 400 is a TrafficControl CR named "web-app-firewall-redirection" and declares all ingress traffic (i.e., based on the direction "In") to Pods matching the label "app=web" in all namespaces (i.e., based on the empty namespaceSelector) to be redirected (i.e., based on the specified action "Redirect") to the device "firewall0" (i.e., based on the attribute value for the device key identifying "firewall0") that performs the L7 packet processing (i.e., an L7 packet processor).

**[0075]** The agent (e.g., agent 122) is responsible for realizing the traffic control request in some embodiments. The agent of some embodiments watches (i.e., has registered for notifications of) the TrafficControl CRs from the API server (e.g., API server 160), and upon being notified of a TrafficControl CR, retrieves attributes of the TrafficControl CR in order to process the attributes to generate flow records (e.g., in the form of OpenFlow rules) for managing container traffic.

**[0076]** As described above, TrafficControl CRs of some embodiments include label selectors that identify Pods and namespaces under which Pods are located. Label selectors do not indicate

or provide uniqueness (i.e., unlike names and UIDs (unique identifiers)), and as such, multiple objects can be associated with the same label. In some embodiments, label selectors allow network administrators to identify a set of objects, and can be based on multiple requirements (e.g., a set of attributes used to define the label selector). The requirements, in some embodiments, are defined according to a set of one or more key/value pairs. For instance, the Pod label selector specified by the TrafficControl CR 400 described above is defined for key: app; value: web. As also described above, the label selectors of some embodiments are left empty in order to match to all (i.e., all Pods or all namespaces).

**[0077]** When the agent receives a notification from the API server of a TrafficControl CR, the agent of some embodiments executes a set of steps. First, the agent uses the label selectors to identify and filter through Pods running on the same node as the agent. For example, based on the TrafficControl CR 400, the agent would identify and select any and all Pods on the same node as the agent that match the label selector "app: web". Once the matching Pods are identified, the agent translates the selected Pods to OVS ports by identifying the OVS ports associated with the selected Pods, and identifying the IP addresses of the identified OVS ports. The agent uses the identified IP addresses of the OVS ports as match attributes in OpenFlow rules for traffic that should be mirrored or redirected to the L7 packet processor. For instance, when one or more Pods 130 are identified and selected based on the label selectors, the agent 122 translates the selected Pods 130 to their associated OVS ports 142 in some embodiments.

**[0078]** After identifying the IP addresses of the OVS ports for the selected Pods, the agent uses the device attribute specified in the TrafficControl CR to identify the L7 packet processor target device to which flows are to be mirrored or redirected. Based on the identified L7 packet processor, the agent identifies an OVS port associated with the L7 packet processor, and uses the address of the identified OVS port for the L7 packet processor as the target port to which the traffic matching the OpenFlow rules should be mirrored or redirected. On a node 105, for example, the agent 122 identifies the port 146 for the L7 packet processor 190 as the target port for matched traffic.

**[0079]** The agent generates flow records using the identified addresses of the OVS ports for the selected Pods, as well as the direction(s) specified in the TrafficControl CR, as match attributes, the action specified by the TrafficControl CR (i.e., redirect or mirror) as the action to perform on flows matching any of the flow records, and the address of the OVS port for the L7 packet

processor as the address to which the matching flows should be redirected or mirrored. The agent then provides the generated flow records to the OVS bridge via the OVS daemons (e.g., the agent 122 provides OpenFlow rules to the OVS bridge 140 via the OVS daemons 124).

**[0080] Figure 5** illustrates an example of calculated OpenFlow rules 500 of some embodiments for first and second Pods, Pod-A and Pod-B, having respective OVS ports veth-a and veth-b. As a result of the OpenFlow rules 500, the data plane no longer transmits traffic toward Pod-A and Pod-B to their OVS ports veth-a and veth-b directly, but rather to the "firewall0" device, from which an L7 packet processing engine will capture traffic. After processing by the L7 packet processing engine, if the decision is "PASS", the traffic will be sent back to the OVS bridge and forwarded to its original destination (e.g., Pod-A via veth-a and/or Pod-B via veth-b).

**[0081]** For embodiments where traffic should be processed in in-line mode (e.g., application layer firewall and IPS), the L7 packet processing engine runs on each node to achieve the least latency. For embodiments that only need a copy of the traffic, the engine can either run on each node or anywhere that is network reachable from the nodes. Additionally, the L7 packet processing engine of some embodiments is capable of capturing traffic from a network device. For features where traffic is redirected, the L7 packet processing engine transmits the accepted traffic via another network device to allow the OVS bridge to continue to forward the accepted traffic to its original destination, according to some embodiments.

**[0082] Figure 6** conceptually illustrates a diagram 600 showing traffic flows between Pods executing on a host that includes an L7 packet processing engine. As shown, the host 610 includes Pods 620 and 625, an OVS bridge 640, an L7 packet processing engine 650, and a port 605. The Pod 620 includes a port, eth0, 630 that connects to a port, vethA, 660 of the OVS bridge 640 for exchanging communications with the Pod 620, while the Pod 625 includes a port, eth0, 635 that connects to a port, vethB, 662 of the OVS bridge 640 for exchanging communications with the Pod 625.

**[0083]** The OVS bridge 640 also includes additional ports s1 664, s2 666, and Tun0 668. When traffic is to be processed by the L7 packet processing engine 650, the traffic is forwarded to the port s1 664 for delivery to the L7 packet processing engine 650. After the processing has been performed, the L7 packet processing engine 650 forwards traffic determined to be allowed back to the OVS bridge 640 via the port s2 666. The port Tun0 668 is a tunnel port used to create overlay tunnels to other nodes (not shown).

**[0084]** For intra-node traffic between the Pods 620 and 625 that does not match any defined TrafficControl CRs, the traffic is forwarded along the path 670. For example, when the Pod 620 sends traffic destined for Pod 625, the traffic is forwarded from port 630 of the Pod 620 to the port 660 of the OVS bridge 640, then to the port 662 of the OVS bridge 640, which forwards the traffic to the port 635 of the Pod 625.

**[0085]** Alternatively, for intra-node traffic between Pods 620 and 625 that does match to a defined TrafficControl CR, the traffic is forwarded along the path 675 to the L7 packet processing engine 650 before delivery to its destination. For example, when the Pod 620 sends traffic matching a TrafficControl CR to the Pod 625, the traffic is forwarded from the port 630 of the Pod 620 to the port 660 of the OVS bridge 640. From the port 660, the traffic is then forwarded to port s1 664 for delivery to the L7 packet processing engine 650. When the traffic is allowed after processing by the L7 packet processing engine 650, the L7 packet processing engine 650 forwards the traffic back to the OVS bridge 640 via the port s2 666. The traffic is then passed to the port 662 where it is delivered to the port 635 of the destination Pod 625.

**[0086]** **Figure 7** conceptually illustrates a diagram 700 showing a traffic flow of some embodiments between Pods on different hosts that both include an L7 packet processing engine. The host 710 includes Pods 720 and 725, an OVS bridge 740, an L7 packet processing engine 750, and a port 764. Similarly, the host 715 includes Pods 730 and 735, an OVS bridge 745, an L7 packet processing engine 755, and a port 774. Each of the Pods 720, 725, 730, and 735 includes a respective port 760, 762, 770, and 772.

**[0087]** The ports 760, 762, 770, and 772 of the Pods 720, 725, 730, and 735 are used to connect the Pods to respective ports of the OVS bridges 740 and 745. For example, port 760 of Pod 720 connects to a port, vethA, 780 of the OVS bridge 740; port 762 of Pod 725 connects to a port, vethB, 782 of the OVS bridge 740; port 770 of Pod 730 connects to a port, vethC, 790 of the OVS bridge 745; and port 772 of Pod 735 connects to a port, vethD, 792 of the OVS bridge 745.

**[0088]** Each OVS bridge 740 and 745 also includes ports for sending and receiving traffic to and from the respective L7 packet processing engines 750 and 755 of their respective hosts 710 and 715. The OVS bridge 740 includes a port s1 784 for redirecting or mirroring traffic to the L7 packet processing engine 750, and a port s2 786 for receiving processed traffic determined to be allowed from the L7 packet processing engine 750. The OVS bridge 745 similarly includes a port s1 796 for redirecting or mirroring traffic to the L7 packet processing engine 755, and a port

s2 798 for receiving processed traffic determined to be allowed from the L7 packet processing engine 755.

**[0089]** To exchange traffic between Pods executing on different hosts, each OVS bridge 740 and 745 includes a respective tunnel port 788 and 794, as shown. As described above, the tunnel ports of some embodiments are created by agents (not shown) deployed to the hosts and are used to create overlay tunnels to other nodes (e.g., between hosts 710 and 715). When traffic is forwarded between the hosts 710 and 715, the traffic traverses an intervening network fabric 799.

**[0090]** In some embodiments, the hosts 710 and 715 are host machines executing on the same physical host computer (not shown), and the intervening network fabric is a software switch on the host computer for connecting the nodes to each other and to network elements external to the host computer. In other embodiments, the hosts 710 and 715 execute on different physical host computers and the intervening network fabric 799 includes a private network (e.g., an MPLS (multiprotocol label switching) network), or includes one or more public networks, such as the Internet and/or one or more networks of one or more public clouds. In still other embodiments, the intervening network fabric 799 includes a combination of public and private networks (such as those mentioned above). The intervening network fabric 799 of some embodiments includes wired or wireless connections, various network forwarding elements (e.g., switches, routers, etc.), etc.

**[0091]** Inter-Pod traffic flows between Pods on the hosts 710 and 715 that do not match TrafficControl CRs are forwarded without being redirected or mirrored to the L7 packet processing engines 750 and 755. For inter-Pod traffic flows between Pods on the hosts 710 and 715 that do match TrafficControl CRs, the traffic flows are redirected or mirrored to L7 packet processing engines 750 and 755 for processing.

**[0092]** For example, a traffic flow from the Pod 720 on host 710 to Pod 735 on host 715 that matches a TrafficControl CR is forwarded along the path 705, which starts from the port 760 of the Pod 720. The port 760 forwards the traffic to the port 780 of the OVS bridge 740, and from the port 780, the traffic is redirected or mirrored to the port s1 784 of the OVS bridge 740 which is an ingress port for the L7 packet processing engine 750. After the traffic has been processed and allowed by the L7 packet processing engine 750, the L7 packet processing engine 750 forwards the allowed traffic back to the OVS bridge 740 via the port s2 786. Alternatively, traffic that is blocked is dropped and not forwarded to the OVS bridge 740.

**[0093]** From the port 786 of the OVS bridge 740, the traffic is then forwarded to the tunnel port 788, which passes the traffic to the port 764 of the host 710. The port 764 forwards the traffic onto the intervening network fabric 799 to the port 744 of the host 715. In some embodiments, the traffic is processed by forwarding elements (e.g., routers, switches, etc.) as it traverses the intervening network fabric 799. Once the traffic is received at the port 774 of the host 715, the port 774 forwards the traffic to the tunnel port 794 of the OVS bridge 745, which redirects or mirrors the traffic to the port s1 796 when the traffic matches an OVS flow rule generated based on a TrafficControl CR so that the traffic can be processed by the L7 packet processing engine 755.

**[0094]** After the traffic has been processed and allowed by the L7 packet processing engine 755, the L7 packet processing engine 755 forwards the traffic back to the OVS bridge 745 via the port s2 798. Traffic that is determined to be blocked is not forwarded, as also mentioned above. The port 798 forwards the traffic to the port 792 of the OVS bridge 745 which corresponds to the Pod 735. The port 792 then delivers the traffic to the port 772 of the destination Pod 735.

**[0095]** **Figure 8** conceptually illustrates a process performed in some embodiments for performing traffic control using a traffic control CR. The process 800 is performed at an OVS bridge of a host machine, in some embodiments, after an agent has performed the set of steps for a TrafficControl CR (e.g., TrafficControl CR 400) described above in order to install OpenFlow rules. The process 800 will be described with references to **Figure 7**.

**[0096]** The process 800 starts when the OVS bridge receives (at 810) a packet associated with a particular Pod on the host machine. For example, the packet of some embodiments can be an ingress packet destined to the particular pod, or an egress packet sent by the particular pod. In some embodiments, the packet is sent from a first Pod on the host machine to a second Pod on the host machine. In other embodiments, the packet is sent from a first Pod on a first host machine to a second Pod on a second host machine.

**[0097]** The process 800 determines (at 820) whether the packet matches any redirection flows created by the network plugin agent (e.g., agent 122). That is, the process 800 determines whether the packet is associated with the pod(s), namespace(s), and flow direction(s) specified by OpenFlow rules installed by a network plugin agent deployed to the host machine. When the packet does not match any redirection flows, the process 800 transitions to 850 to forward the packet toward a destination of the packet. When the packet does match at least one redirection

flow, the process 800 forwards (at 830) the packet to a first port associated with the packet processing device.

**[0098]** For example, in the diagram 700, a packet flow sent from the Pod 720 on host 710 to the Pod 735 on host 715 is forwarded from the port 780 that is associated with the Pod 720 to the port s1 784 to be processed by the L7 packet processing engine 750. Similarly, on the destination side at host 715, the packet flow is forwarded to the L7 packet processing engine 755 of the host 715 before the flow is forwarded to the destination Pod 735.

**[0099]** The process 800 receives (at 840) the processed packet from the packet processing device via a second port associated with the packet processing device. That is, when the packet is allowed following processing by the packet processing device, the packet processing device provides the packet back to the OVS bridge for forwarding to its destination either on the same host as where the processing occurred or on a different host. When the packet is blocked following processing by the packet processing device, the packet is dropped, in some embodiments, and not passed back to the OVS bridge.

**[00100]** The process forwards (at 850) the packet toward a destination of the packet. For example, after the L7 packet processing engine 750 processes packets sent by the Pod 720, the L7 packet processing engine 750 passes the packets back to the OVS bridge 740, which forwards the packets via the tunnel port 788 to the port 764 of the host 710 for forwarding to the host 715. On the host 715, after the OVS bridge 745 receives processed packets from the L7 packet processing engine 755 belonging to the flow between Pod 720 and Pod 735, the OVS bridge 745 forwards those packets to the Pod 735 via the port 792 of the OVS bridge 745, as illustrated by path 705. Following 850, the process 800 ends.

**[00101]** Although the above-described embodiments refer to an L7 packet processor as the device that receives directed flows based on the flow records generated from the TC CRs, in other embodiments the TC CR instances are used to generate flow records that direct flows to other types of packet processors, e.g., to L3/L4 middlebox services, to forwarding elements (e.g., switches, routers, gateways, etc.), to third party service appliances, etc.

**[00102]** Many of the above-described features and applications are implemented as software processes that are specified as a set of instructions recorded on a computer-readable storage medium (also referred to as computer-readable medium). When these instructions are executed by one or more processing unit(s) (e.g., one or more processors, cores of processors, or

other processing units), they cause the processing unit(s) to perform the actions indicated in the instructions. Examples of computer-readable media include, but are not limited to, CD-ROMs, flash drives, RAM chips, hard drives, EPROMs, etc. The computer-readable media does not include carrier waves and electronic signals passing wirelessly or over wired connections.

**[00103]** In this specification, the term "software" is meant to include firmware residing in read-only memory or applications stored in magnetic storage, which can be read into memory for processing by a processor. Also, in some embodiments, multiple software inventions can be implemented as sub-parts of a larger program while remaining distinct software inventions. In some embodiments, multiple software inventions can also be implemented as separate programs. Finally, any combination of separate programs that together implement a software invention described here is within the scope of the invention. In some embodiments, the software programs, when installed to operate on one or more electronic systems, define one or more specific machine implementations that execute and perform the operations of the software programs.

**[00104]** **Figure 9** conceptually illustrates a computer system 900 with which some embodiments of the invention are implemented. The computer system 900 can be used to implement any of the above-described hosts, controllers, gateway, and edge forwarding elements. As such, it can be used to execute any of the above described processes. This computer system 900 includes various types of non-transitory machine-readable media and interfaces for various other types of machine-readable media. Computer system 900 includes a bus 905, processing unit(s) 910, a system memory 925, a read-only memory 930, a permanent storage device 935, input devices 940, and output devices 945.

**[00105]** The bus 905 collectively represents all system, peripheral, and chipset buses that communicatively connect the numerous internal devices of the computer system 900. For instance, the bus 905 communicatively connects the processing unit(s) 910 with the read-only memory 930, the system memory 925, and the permanent storage device 935.

**[00106]** From these various memory units, the processing unit(s) 910 retrieve instructions to execute and data to process in order to execute the processes of the invention. The processing unit(s) 910 may be a single processor or a multi-core processor in different embodiments. The read-only-memory (ROM) 930 stores static data and instructions that are needed by the processing unit(s) 910 and other modules of the computer system 900. The permanent storage device 935, on the other hand, is a read-and-write memory device. This device 935 is a non-

volatile memory unit that stores instructions and data even when the computer system 900 is off. Some embodiments of the invention use a mass-storage device (such as a magnetic or optical disk and its corresponding disk drive) as the permanent storage device 935.

**[00107]** Other embodiments use a removable storage device (such as a floppy disk, flash drive, etc.) as the permanent storage device. Like the permanent storage device 935, the system memory 925 is a read-and-write memory device. However, unlike storage device 935, the system memory 925 is a volatile read-and-write memory, such as random access memory. The system memory 925 stores some of the instructions and data that the processor needs at runtime. In some embodiments, the invention's processes are stored in the system memory 925, the permanent storage device 935, and/or the read-only memory 930. From these various memory units, the processing unit(s) 910 retrieve instructions to execute and data to process in order to execute the processes of some embodiments.

**[00108]** The bus 905 also connects to the input and output devices 940 and 945. The input devices 940 enable the user to communicate information and select commands to the computer system 900. The input devices 940 include alphanumeric keyboards and pointing devices (also called "cursor control devices"). The output devices 945 display images generated by the computer system 900. The output devices 945 include printers and display devices, such as cathode ray tubes (CRT) or liquid crystal displays (LCD). Some embodiments include devices such as touchscreens that function as both input and output devices 940 and 945.

**[00109]** Finally, as shown in **Figure 9**, bus 905 also couples computer system 900 to a network 965 through a network adapter (not shown). In this manner, the computer 900 can be a part of a network of computers (such as a local area network ("LAN"), a wide area network ("WAN"), or an Intranet), or a network of networks (such as the Internet). Any or all components of computer system 900 may be used in conjunction with the invention.

**[00110]** Some embodiments include electronic components, such as microprocessors, storage and memory that store computer program instructions in a machine-readable or computer-readable medium (alternatively referred to as computer-readable storage media, machine-readable media, or machine-readable storage media). Some examples of such computer-readable media include RAM, ROM, read-only compact discs (CD-ROM), recordable compact discs (CD-R), rewritable compact discs (CD-RW), read-only digital versatile discs (e.g., DVD-ROM, dual-layer DVD-ROM), a variety of recordable/rewritable DVDs (e.g., DVD-RAM,

DVD-RW, DVD+RW, etc.), flash memory (e.g., SD cards, mini-SD cards, micro-SD cards, etc.), magnetic and/or solid state hard drives, read-only and recordable Blu-Ray® discs, ultra-density optical discs, any other optical or magnetic media, and floppy disks. The computer-readable media may store a computer program that is executable by at least one processing unit and includes sets of instructions for performing various operations. Examples of computer programs or computer code include machine code, such as is produced by a compiler, and files including higher-level code that are executed by a computer, an electronic component, or a microprocessor using an interpreter.

**[00111]** While the above discussion primarily refers to microprocessor or multi-core processors that execute software, some embodiments are performed by one or more integrated circuits, such as application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs). In some embodiments, such integrated circuits execute instructions that are stored on the circuit itself.

**[00112]** As used in this specification, the terms "computer", "server", "processor", and "memory" all refer to electronic or other technological devices. These terms exclude people or groups of people. For the purposes of the specification, the terms "display" or "displaying" mean displaying on an electronic device. As used in this specification, the terms "computer-readable medium," "computer-readable media," and "machine-readable medium" are entirely restricted to tangible, physical objects that store information in a form that is readable by a computer. These terms exclude any wireless signals, wired download signals, and any other ephemeral or transitory signals.

**[00113]** While the invention has been described with reference to numerous specific details, one of ordinary skill in the art will recognize that the invention can be embodied in other specific forms without departing from the spirit of the invention. Thus, one of ordinary skill in the art would understand that the invention is not to be limited by the foregoing illustrative details, but rather is to be defined by the appended claims.

## CLAIMS

1. A method of performing layer 7 (L7) packet processing for a set of Pods executing on a host computer, the set of Pods managed by a container orchestration platform, the method comprising:

at the host computer:

receiving notification of a creation of a traffic control (TC) custom resource (CR) that is defined by reference to a TC custom resource definition (CRD);

identifying a set of interfaces of a set of one or more managed forwarding elements (MFEs) executing on the host computer that are candidate interfaces for receiving flows that need to be directed based on the TC CR to a layer 7 packet processor; and

based on the identified set of interfaces, providing a set of flow records to the set of MFEs to process in order to direct a subset of flows that the set of MFEs receive to the layer 7 packet processor.

2. The method of claim 1, wherein the notification is received from the container orchestration platform, which created the TC CR that references the TC CRD.

3. The method of clam 2, wherein an API server of the container orchestration platform creates the TC CR after receiving an API directing the API server to create the TC CR, and then parsing the API to extract parameters used to define the TC CR.

4. The method of claim 1, wherein the identified set of interfaces comprises an interface associated with a Pod that is a source of a flow that needs to be directed to the L7 packet processor.

5. The method of claim 1, wherein the identified set of interfaces comprises an interface associated with a Pod that is a destination of a flow that needs to be directed to the L7 packet processor.

6. The method of claim 1 further comprising

from the TC CR, identifying the L7 packet processor;

identifying an MFE interface associated with the L7 packet processor; and

using the identified MFE interface associated with the L7 packet processor in the set of flow records.

7. The method of claim 1, wherein the L7 packet processor executes on the host computer.

8. The method of claim 1, wherein the L7 packet processor executes outside of the host computer.

9. The method of claim 1, wherein the L7 packet processor performs a middlebox service operation based on L7 parameters extracted from flows processed by the L7 packet processor.

10. The method of claim 9, wherein the L7 packet processor performs deep packet inspection (DPI) to extract L7 parameters from the flows.

11. The method of claim 10, wherein middlebox service operations comprise at least one of firewall operations, load balancing operations, network address translation operation, intrusion detection operations, intrusion prevention operations.

12. The method of claim 1, wherein the TC CR uses high level attributes to define source or destination Pods associated with the flows that are to be directed to the L7 packet processor, and the set of flow records specifies interfaces of the source or destination Pods in terms of network addresses associated with the source or destination Pods.

13. The method of claim 1 further comprising generating, on the host computer, the flow records by processing attributes specified by the TC CR.

14. The method of claim 1, wherein at least one particular flow is directed to the L7 packet processor by redirecting the particular flow to the L7 packet processor.

15. The method of claim 1, wherein at least one particular flow is directed to the L7 packet processor through a mirroring operation that copies the particular flow and sends the copy to the L7 packet processor.

16. A non-transitory machine readable medium storing a program for execution by a set of processing units of a host computer, the program for performing layer 7 (L7) packet processing for a set of Pods executing on the host computer, the set of Pods managed by a container orchestration platform, the program comprising sets of instructions for:

receiving notification of a creation of a traffic control (TC) custom resource (CR) that is defined by reference to a TC custom resource definition (CRD);

identifying a set of interfaces of a set of one or more managed forwarding elements (MFEs) executing on the host computer that are candidate interfaces for receiving flows that need to be directed based on the TC CR to a layer 7 packet processor; and

based on the identified set of interfaces, providing a set of flow records to the set of MFEs to process in order to direct a subset of flows that the set of MFEs receive to the layer 7 packet processor.

17. The non-transitory machine readable medium of claim 16, wherein:

the notification is received from the container orchestration platform; and

an API server of the container orchestration platform creates the TC CR after (i) receiving an API directing the API server to create the TC CR, and then (ii) parsing the API to extract parameters used to define the TC CR.

18. The non-transitory machine readable medium of claim 16, wherein the identified set of interfaces comprises an interface associated with a Pod that is a source of a flow that needs to be directed to the L7 packet processor.

19. The non-transitory machine readable medium of claim 16, wherein the identified set of interfaces comprises an interface associated with a Pod that is a destination of a flow that needs to be directed to the L7 packet processor.

20. The non-transitory machine readable medium of claim 16 further comprising sets of instructions for:

from the TC CR, identifying the L7 packet processor;

identifying an MFE interface associated with the L7 packet processor; and

using the identified MFE interface associated with the L7 packet processor in the set of flow records.

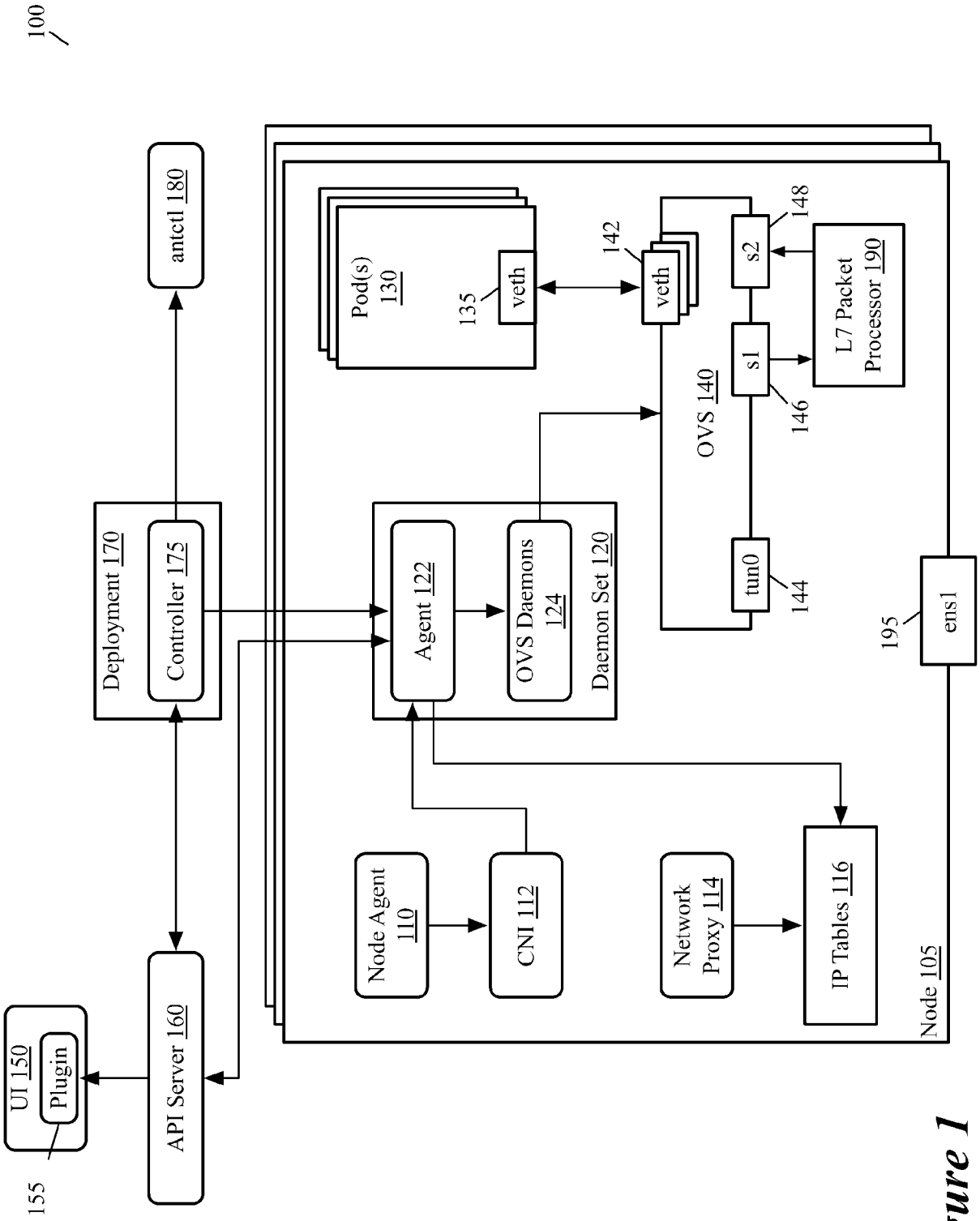
100
UI 150
Plugin
155
API Server 160
Deployment 170
Controller 175
antctl 180
Node 105
Pod(s) 130
135
veth
142
veth
OVS 140
s2
148
s1
146
L7 Packet Processor 190
tun0
144
195
ens1
Agent 122
OVS Daemons 124
Daemon Set 120
Node Agent 110
CNI 112
Network Proxy 114
IP Tables 116

*Figure 1*

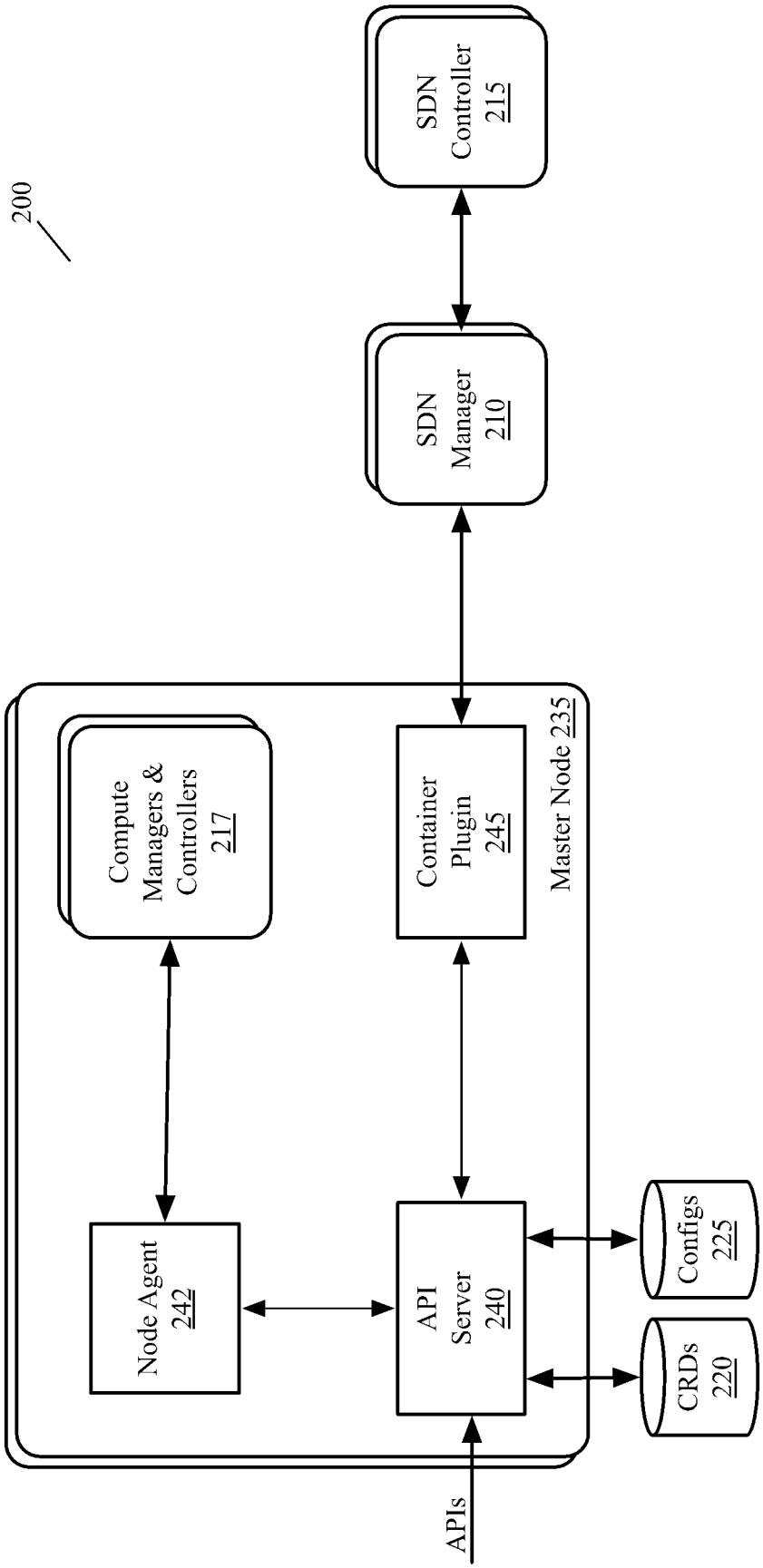
200
SDN Controller 215
SDN Manager 210
Compute Managers & Controllers 217
Container Plugin 245
Master Node 235
Node Agent 242
API Server 240
Configs 225
CRDs 220
APIs

*Figure 2*

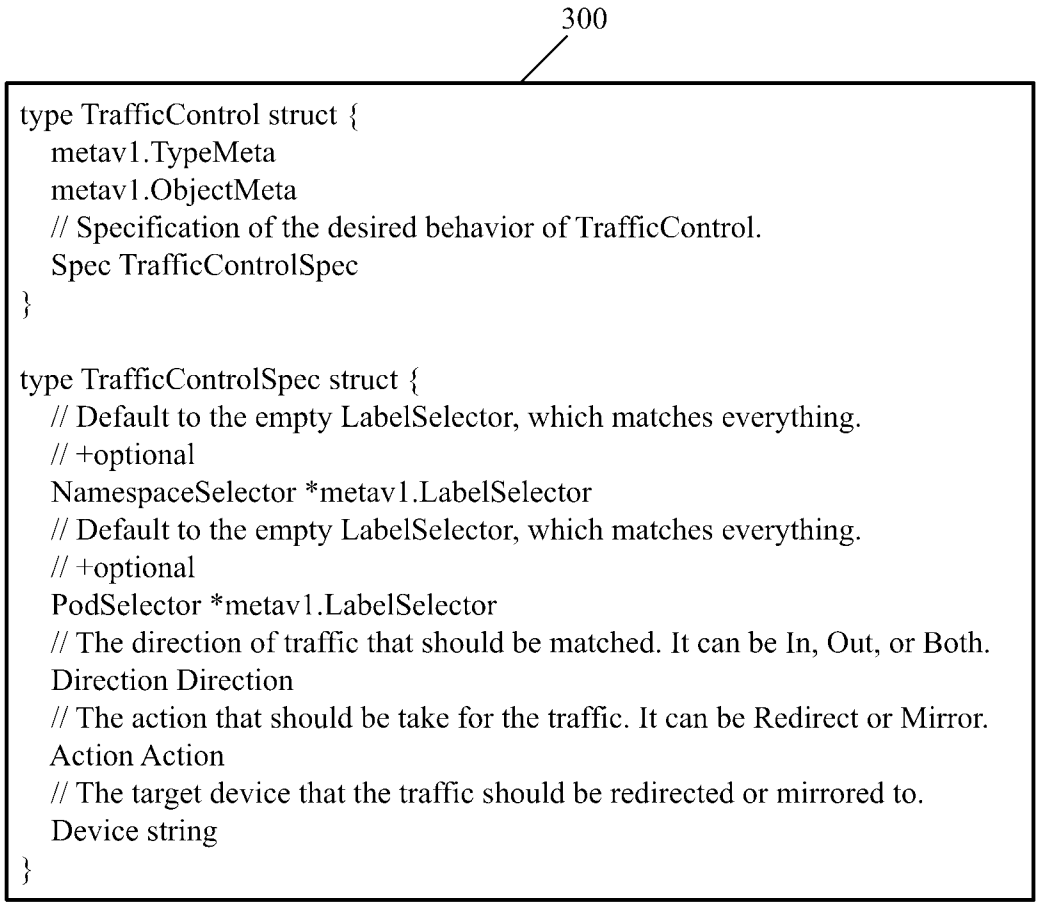
300
type TrafficControl struct {
metav1.TypeMeta
metav1.ObjectMeta
// Specification of the desired behavior of TrafficControl.
Spec TrafficControlSpec
}
type TrafficControlSpec struct {
// Default to the empty LabelSelector, which matches everything.
// +optional
NamespaceSelector *metav1.LabelSelector
// Default to the empty LabelSelector, which matches everything.
// +optional
PodSelector *metav1.LabelSelector
// The direction of traffic that should be matched. It can be In, Out, or Both.
Direction Direction
// The action that should be take for the traffic. It can be Redirect or Mirror.
Action Action
// The target device that the traffic should be redirected or mirrored to.
Device string
}

***Figure 3***

400

```
apiVersion: crd.antrea.io/v1alpha1
kind: TrafficControl
metadata:
  name: web-app-firewall-redirection
spec:
  podSelector:
    matchLabels:
      app: web
    namespaceSelector: {}
    direction: In
    action: Redirect
    device: firewall0
```

*Figure 4*

500

```
table=TrafficControl, ct_state=+new+trk,ip,
actset_output="veth-a" actions=load:0x1->NXM_NX_REG2[0]
table=TrafficControl, ct_state=+new+trk,ip,
actset_output="veth-b" actions=load:0x1->NXM_NX_REG2[0]
table=ConntrackCommit, reg2=0x1/0x1
actions=ct(commit,table=HairpinSNAT,zone=65520,exec(load:0x24-
>NXM_NX_CT_MARK[]))
table=Output, ct_state=+trk,ct_mark=0x24
actions=output:"firewall0"
```

*Figure 5*

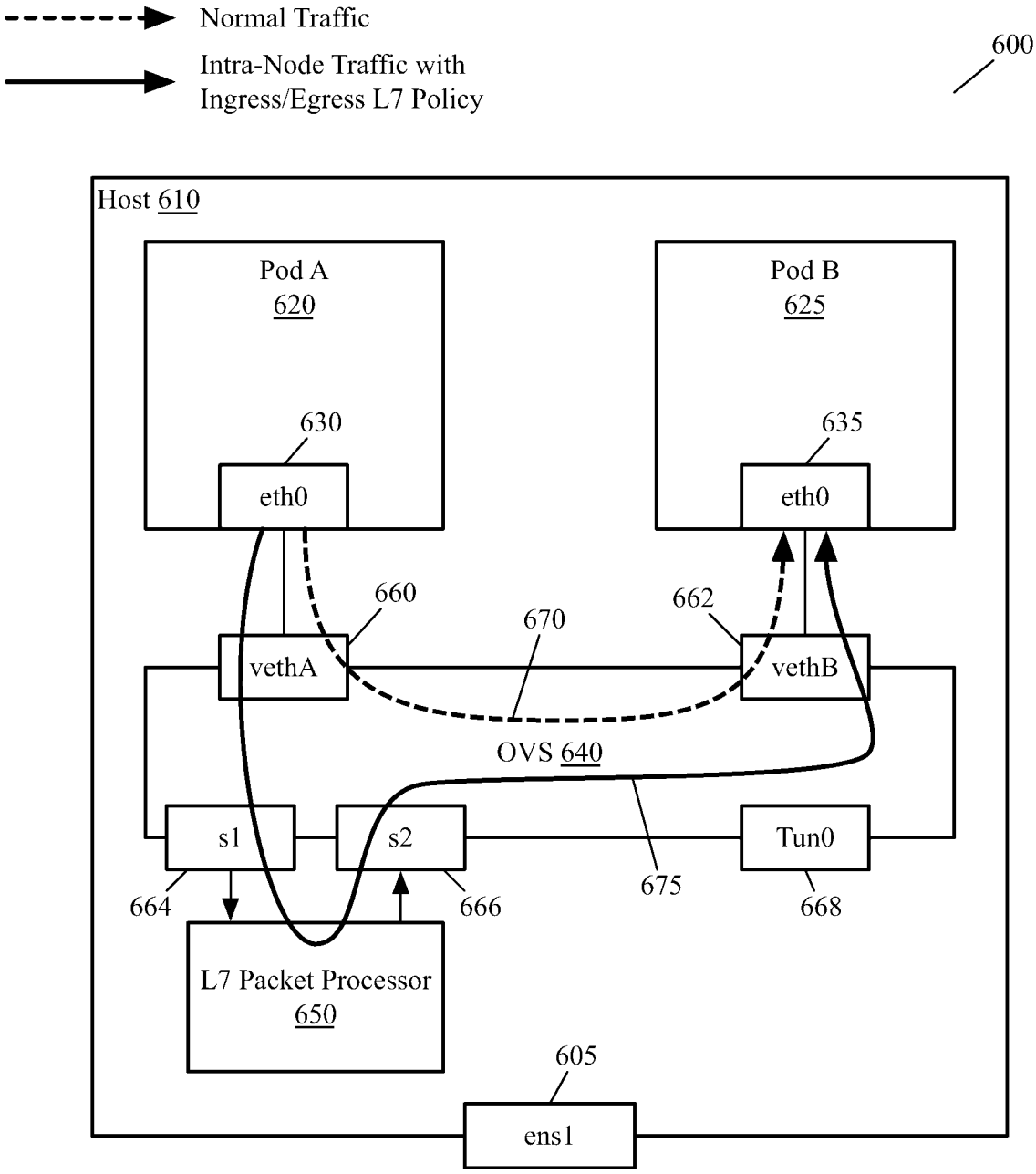
Normal Traffic
Intra-Node Traffic with Ingress/Egress L7 Policy
600
Host 610
Pod A 620
Pod B 625
630
635
eth0
eth0
660
662
670
vethA
vethB
OVS 640
s1
s2
Tun0
664
666
675
668
L7 Packet Processor 650
605
ens1

***Figure 6***

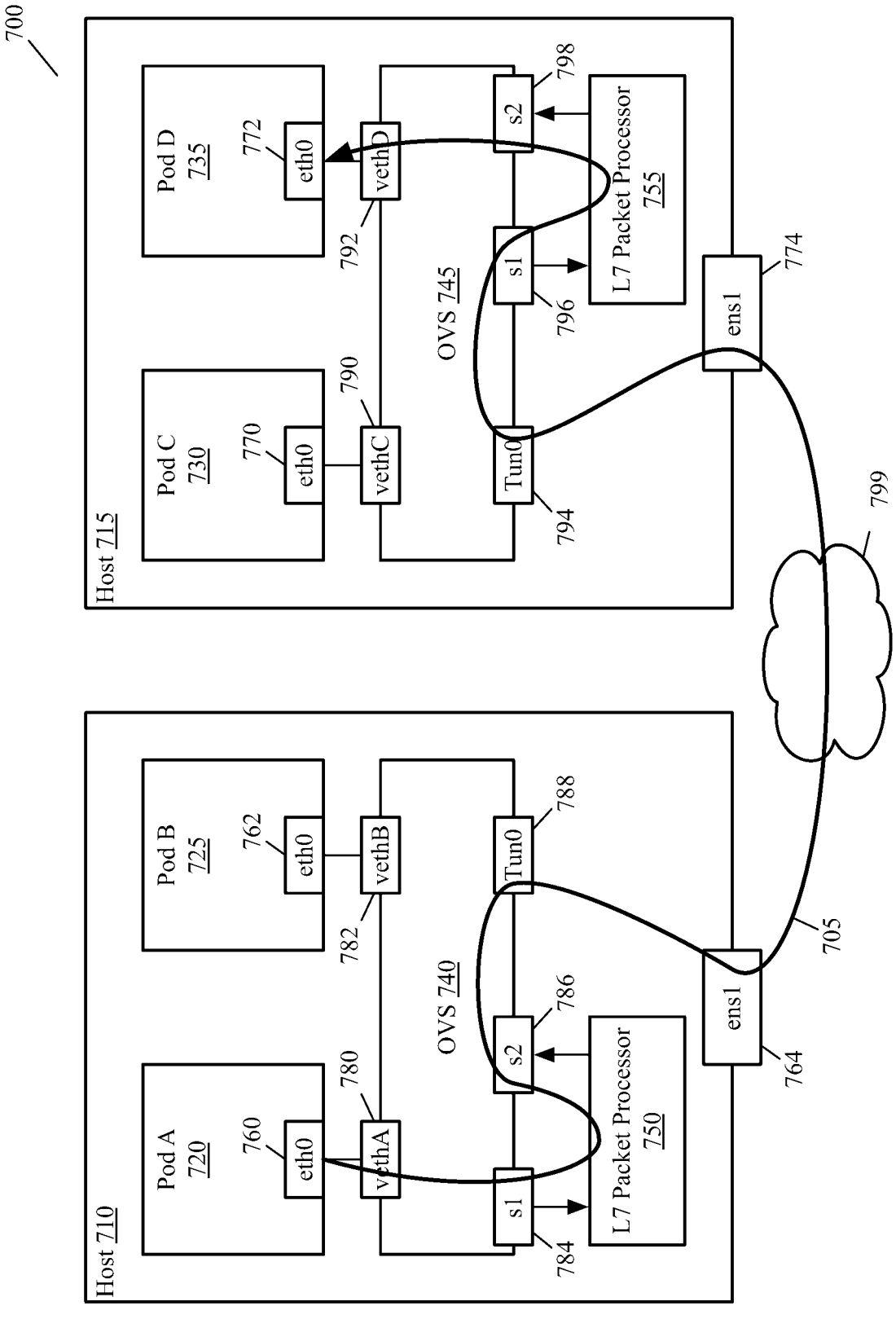
700
Host 710
Pod A 720
760
eth0
780
vethA
Pod B 725
762
eth0
782
vethB
OVS 740
s1
784
s2
786
Tun0
788
L7 Packet Processor 750
ens1
764
705
799
Host 715
Pod C 730
770
eth0
790
vethC
Pod D 735
772
eth0
792
vethD
OVS 745
s1
796
s2
798
Tun0
794
L7 Packet Processor 755
ens1
774
Inter-Node Traffic with Both Ingress and Egress L7 Policy

*Figure 7*

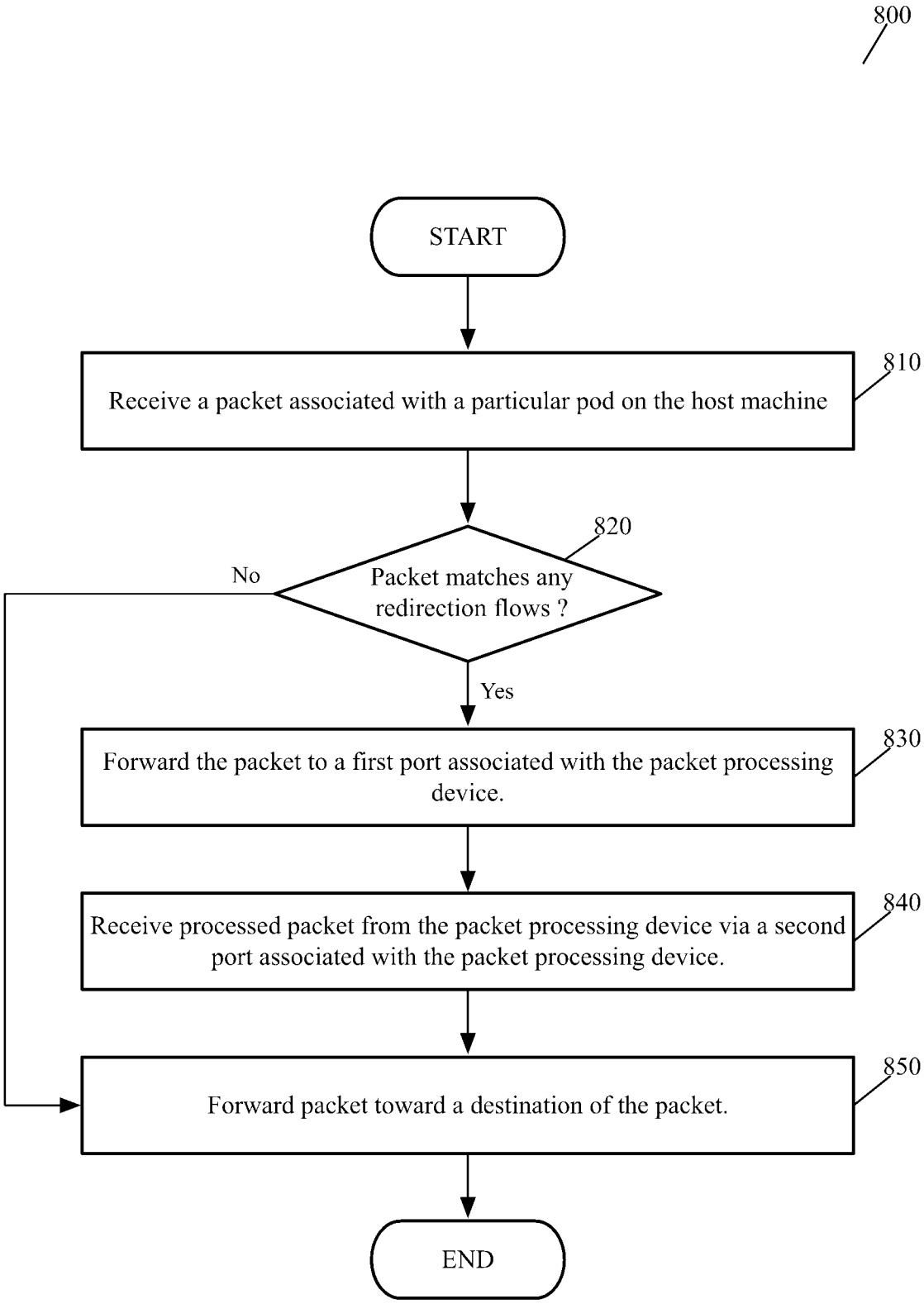
800
START
Receive a packet associated with a particular pod on the host machine
810
820
Packet matches any redirection flows ?
No
Yes
Forward the packet to a first port associated with the packet processing device.
830
Receive processed packet from the packet processing device via a second port associated with the packet processing device.
840
Forward packet toward a destination of the packet.
850
END

***Figure 8***

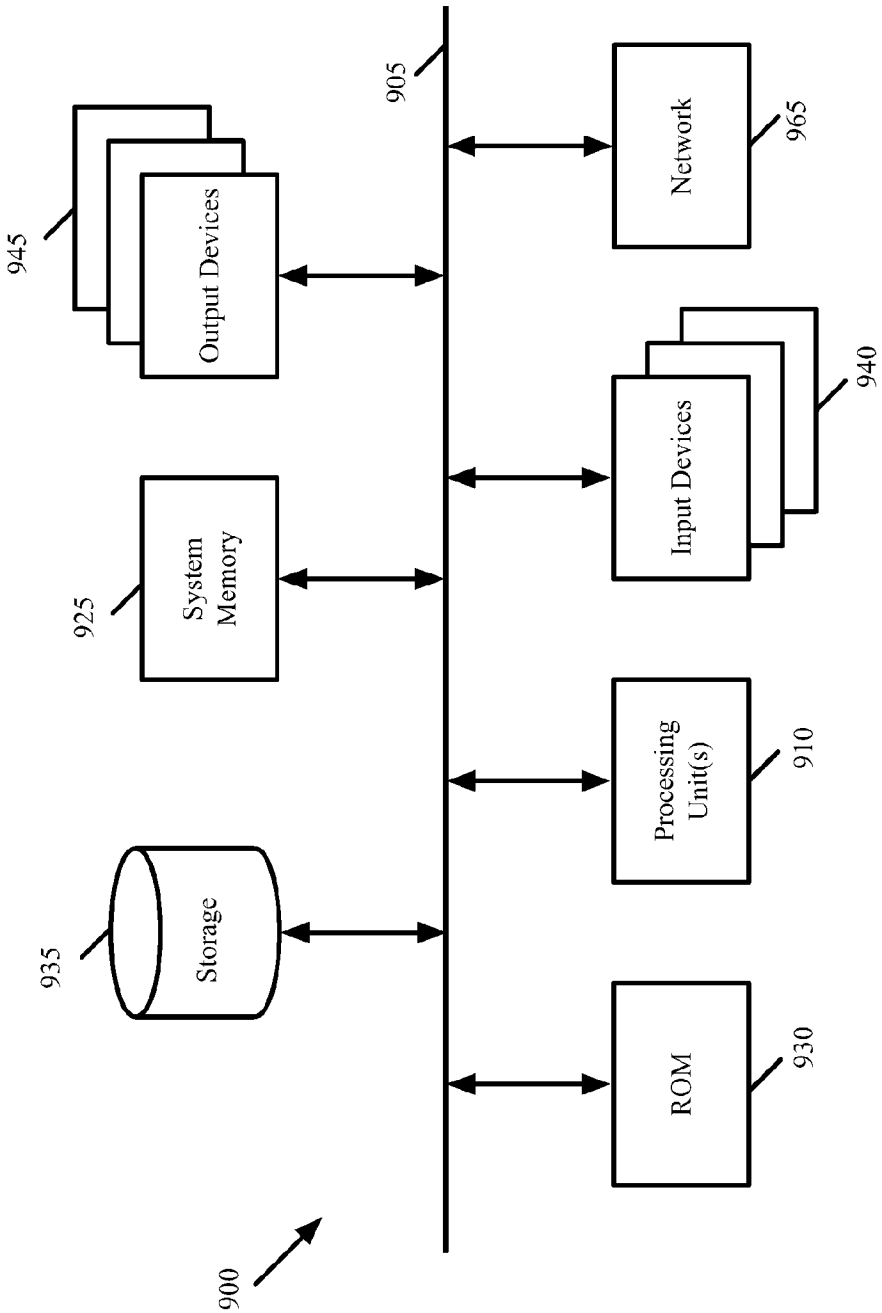
900
905
945
Output Devices
925
System Memory
935
Storage
Network
965
Input Devices
940
Processing Unit(s)
910
ROM
930

*Figure 9*

# INTERNATIONAL SEARCH REPORT

International application No
PCT/CN2023/099675

**A. CLASSIFICATION OF SUBJECT MATTER**
INV. G06F9/455 H04L41/08 H04L41/0895
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)
G06F H04L

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal, WPI Data

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| X | Palomero Álvarez Pablo ET AL: "A scalable platform for traffic analysis on commodity hardware", , 16 July 2018 (2018-07-16), pages 1-114, XP093099647, Retrieved from the Internet: URL:https://oa.upm.es/51641/1/PFC_PABLO_PALOMERO_ALVAREZ.pdf [retrieved on 2023-11-09] page 12 - page 89 ----- -/-- | 1-20 |

[x] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance;; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance;; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

| Date of the actual completion of the international search | Date of mailing of the international search report |
|---|---|
| 9 November 2023 | 21/11/2023 |

| Name and mailing address of the ISA/ | Authorized officer |
|---|---|
| European Patent Office, P.B. 5818 Patentlaan 2<br>NL - 2280 HV Rijswijk<br>Tel. (+31-70) 340-2040,<br>Fax: (+31-70) 340-3016 | Kalejs, Eriks |

1

Form PCT/ISA/210 (second sheet) (April 2005)

# INTERNATIONAL SEARCH REPORT

International application No
PCT/CN2023/099675

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| A | SHIXIONG QI ET AL: "MiddleNet: A Unified, High-Performance NFV and Middlebox Framework with eBPF and DPDK", ARXIV.ORG, CORNELL UNIVERSITY LIBRARY, 201 OLIN LIBRARY CORNELL UNIVERSITY ITHACA, NY 14853, 8 March 2023 (2023-03-08), XP091457723, page 3 - page 5 | 1-20 |

1