Why and How to Use Contrail APIs

Designed to increase business innovation, Contrail APIs abstract the network by interoperating with open cloud orchestration systems.

Table of Contents

List of Figures

Executive Summary

Juniper Networks started opening up the Juniper Networks® Junos® operating system via APIs in the late 1990s. The first implementation was done via undocumented APIs based on JavaScript. In 2001, after the APIs were rewritten to use XML, those APIs were published and made widely available as the Junos OS XML management protocol. The IETF standardized Junos OS XML management protocol, which became what is today known as NETCONF. The Junos OS XML management protocol became very popular among the Juniper customer base, as it enabled customers to write automation tools natively on the network platform that were not based on screen scraping.

In the mid 2000s, following the success of the Junos OS XML management protocol, Juniper Networks Junos SDK program was started and provided APIs to third-party developers for the control and service planes. The APIs were low level (C/C++ libraries) and enabled customers to add new control and service applications to network platforms. Released in 2010, Junos Space SDK provided APIs for the management plane. All programs had advantages and disadvantages, but they provided an invaluable step forward in network automation and tooling. Juniper is using what has been learned through the development and use of these various programming interfaces to bring a more refined solution to the industry and market.

One of the new concepts that has created a lot of buzz in the industry is software-defined networking (SDN), which brings new concepts of programmability to network operators. SDN explicitly defines four planes in networks:

- Data or forwarding
- Control
- Service
- Management

Each plane provides APIs that allow faster introduction of new features and services. It is much easier to test and innovate new services and features using software than wait for equipment manufacturers to add needed features so that network operators can satisfy their customer requests. Some of the requests can't even be answered, as the cost of doing some custom work for a single network operator can be prohibitive. Integration with OSS and BSS becomes easier as well, as all systems are software based.

Introduction

Over the last 30 years, compute hardware and software have advanced rapidly, but the way that networking is done has remained virtually unchanged. The basic networking concepts are still the same as in the 1970s. At the same time, while networks have become a critical component in all parts of daily life, traditional networking approaches have become too complex, closed, and proprietary. In fact, they often represent a barrier to faster innovation and services that address the new demands of consumer and enterprise environments.

The networks of today have become:

- **Complex**—They are difficult to manage, scale, design, and to understand and analyze.
- **Slow**—Many operations are still manual, too static, and too slow to respond to customer requests.
- **Closed**—They are not programmable (do not mix automation and programmability) or accessible to third parties for innovation and monetization. They are also monolithic and inflexible.
- **Uneconomical**—IT CapEx and OpEx costs are increasing with rising complexity and as they are closed, it is hard to differentiate and monetize.
- **Unreliable**—The biggest problem is the human error in today's network operation. Networks are complex and many operations are still manual, which means error margins are high due to human interventions.

When carriers want to deploy a new feature, it is usually tied to a new box, which requires a change in the physical design of the network through a slow process. At the end, integration with network IT departments ends up being another complex, time-consuming, and expensive process.

Virtualizing the Network

There have been attempts in the past to provide programmable interfaces to the networking platform (SNMP, CAMEL, IN, just to mention a few), but they have all been complex and proprietary, and many providers created their own solutions which were not providing separation of networking planes.

Juniper Networks Contrail is a network and services virtualization system designed to increase business innovation, improve system-level orchestration, and decrease networking costs. It works within open cloud orchestration systems such as OpenStack and CloudStack. Contrail includes an open, standards-based SDN controller that virtualizes the network to enable automation and orchestration of hybrid cloud environments, elastic service chaining of network and security services, and a robust "Big Data for Infrastructure" (BDI) analytics engine that provides a real-time view of the entire network.

The Contrail solution also provides APIs, which enable manipulation of configuration elements exposed by the Virtual Network Server (VNS), as well as analytics and operational information. Interaction with VNS is possible through the REST interface or Python, Java, and JavaScript libraries.

Why Should You Use APIs?

By using Contrail APIs, you make certain that your applications are more robust. As the implementation is hidden and exposed only via an API, Juniper Networks can improve implementation and add new enhancements without causing disruption to the existing applications using that API. New enhancements can be provided automatically to the existing applications. The API can support multiple versions of the Contrail solution and detect which version it needs to use at runtime. Applications developed by network operators and third parties have increased longevity and are less fragile, rigid, and immobile. Over a longer time, this brings financial savings, as the applications can be easily extended without the need to rewrite them from scratch.

Applications can have a modular design when APIs are used. APIs normally address a specific task or use case. As such, APIs usually define a modular grouping of functionality with a coherent focus. Developing an application on top of a collection of APIs promotes loosely coupled and modular architectures where the behavior of one module is not dependent on the internal details of another module. Another benefit for that is reusability of modules for different applications.

Why Python and REST?

Today, speed of development, ease of maintenance, and availability of different libraries are as important as ever, and new developer tools are coming to market that answer those requirements better than before. Python, although available since late 80s, is the new scripting language that is taking the place held previously by Perl. The ease with which a programmer of other languages can pick up basic Python skills, and the huge standard library, are key to Python adoption. Any network project of size has tasks to automate, and automating them in Python is orders of magnitude faster than using more mainstream languages. Tools can be created to extract information from the network and present them in a way that is useful to other applications and systems. Python provides for quick implementation, ease of reading the code, and low maintenance—all of which make it easy for other developers to continue from where the last developer finished. Even people without a background in Python can patch the applications without breaking them.

Juniper understands well that not all development environments are set up to support Python and that these environments are using other programming languages. Representational State Transfer (REST) answers this requirement that Contrail's VNS can be accessible from any programming language as long as that language provides an HTTP library.

REST is a design idiom that employs the Web's stateless client/server architecture to represent REST Web services as resources identified by a URL. This is also the tiniest possible description of what REST actually means. It is not a standard, rather a style describing the act of transferring a state of something by its representation. REST dramatically reduces the complexity of communication by making the representation the only contract (compare that to RPCs). We don't want our communication to fail because two communicating sides understand a procedure differently.

Another benefit is that with REST, sessions are not needed. Anything can be done without sessions. Information is encapsulated in the request of the URI, or even another resource can be created—for example, virtual network is "/virtual-network/1234."

Between two requests, the server can be turned off, the platform and OS uninstalled and then reinstalled, and the platform and application restored from the backup—all without the client detecting any of this. There is no state stored in the browser, so only cookies have to be taken care of.

Unlock the Power of Network

Networks can be viewed as closed loop systems. Typical actions that are taken to affect network assets include permanent actions such as configuration of network devices and provisioning of services in response to customer orders. There could also be dynamic actions such as quality-of-service (QoS) policy provisioning, enforcing security policies, etc. This is classic network management, and there is nothing special to that.

Network platforms of today lack the ability to marry the action and asset side of the cycle based on the data that is generated by the network assets, and that is a major pain point for developers. Network assets generate a lot of data. All this data by itself is worthless unless you can turn it into value by understanding what it means in the business context and what it might do operationally for your customers or your company.

What Is the Solution?

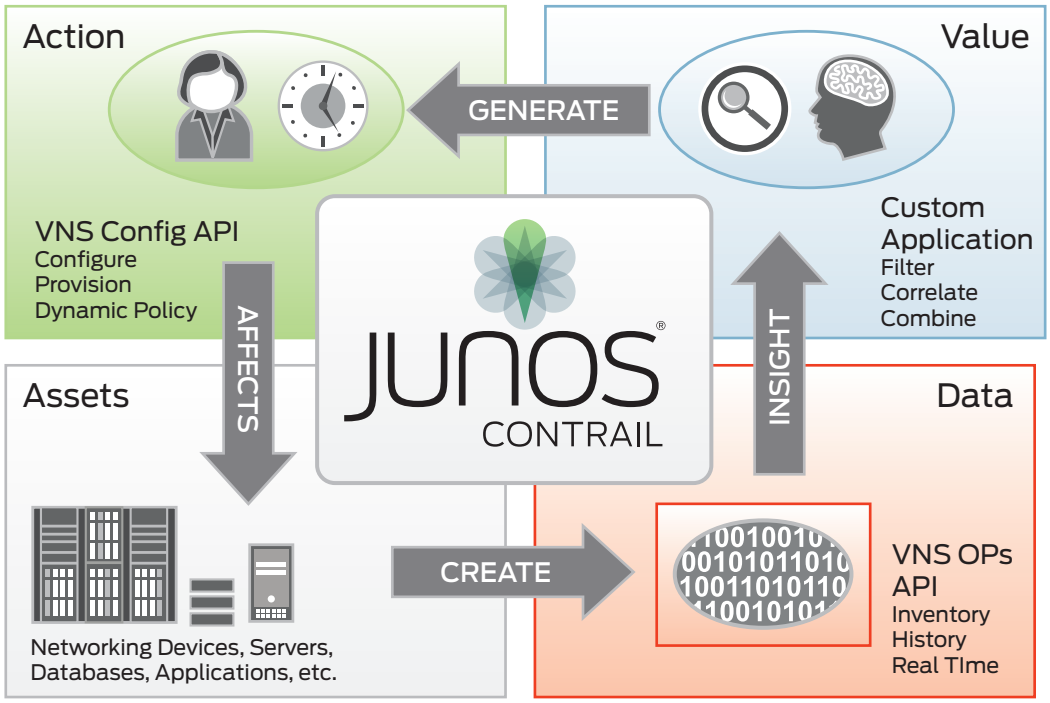

Figure 1: Network as a Closed Loop System

Using Contrail APIs changes the equation. Referring to Figure 1, Contrail marries the Action and Assets side of the cycle on the left with the Data and Value side on the right. It provides the ability to suck in the static and dynamic behavior of the network and analyze this behavior in the context that can be provided by an internal or an external source. The internal source could be an event on the network itself, and the external source could be some business intelligence or a truly external entity in the form of user behavior, for example. Contrail combines the ability to access the static and the dynamic nature of networks and the wealth of information hidden in them with the ability to combine this data with an internal or an external source. This combined with a highly scalable and resilient platform, and a great set of tools that allows developers to easily utilize the wealth of information, makes Contrail a great platform to develop new, next-generation applications.

Contrail provides APIs for the following modules:

Operational module—The ops server provides APIs to get the current state of the User Visible Entities in the Contrail VNS. User Visible Entity (UVE) is defined as an object that can span multiple components and require aggregation before the UVE information is presented.

Virtual Network: This UVE provides information associated with a virtual network such as:

- List of networks connected to this network
- List of virtual machines spawned in this virtual network
- List of access control lists (ACLs) associated with this virtual network
- Global input/output statistics
- Per virtual network pair of input/output statistics

The REST API to get this UVE is through HTTP GET using the URL /analytics/virtual-network/<name>.

Virtual Machine (VM): This UVE provides information associated with a virtual machine such as:

- List of interfaces in this VM
- List of floating IPs associated with each of its interfaces
- Input/output statistics

The REST API to get this UVE is through HTTP GET using the URL /analytics/virtual-machine/<name>.

VRouter: This UVE provides information associated with a VRouter such as:

- Configuration module—The Juniper Networks Contrail configuration API server allows for manipulation of configuration elements exposed by the VNS. Interaction with this server is possible through either a REST interface or by a Python library.

The following UVEs can be manipulated through the APIs:

- Virtual Networks (L2/L3)—Create/Read/Update/Delete
- Virtual Network Policies—Create/Read/Update/Delete
  - Security, QoS, IPAM Rules, Floating IP, Analyzer/Mirroring

Both Python and REST APIs are designed so that they can be used from anywhere. The Python library is designed on top of REST, so by using the Python library and importing the library into your application, REST communicates to the VNS configuration and ops servers. This is achieved by specifying the contrail.conf file in the configuration directory, which contains the server IP address and port number.

Here is an example of how to specify connectivity to the configuration server. By default, the server uses port 8082, and the ops server uses port 8081.

```
[DEFAULTS]
api_server_ip=10.84.18.11
api_server_port=8082
ops_server_ip=10.84.18.11
ops_server_port=8081
```

Once connectivity is defined, the Python library determines where to connect, and data can be exchanged between the application and VNS.

There are many REST clients—some run from the browser, and some run from Integrated Development Environment (IDE) applications. Both have pluses and minuses. This document does not go into details on REST client specifics. The good news is that all can be used. For users who are comfortable with CLI, the UNIX utility curl can also be used for fast prototyping and debugging.

Once workflow is created, REST calls and data structure can be defined. It is also easy to verify by using a REST client if the correct data is being sent and received. Once the calls are verified, those can be placed into any programming language, where the data is manipulated into the correct format by the application logic, instead of manually.

In order to be able to create new services, the following items have to be already available, set up, or both:

- **Images or snapshots of virtual machines**—New instances can be created from images or snapshots. These should be listed in the images, snapshot folder, or both of the service orchestration system.
- **Virtual networks**—These are used to connect instances to VRouter. Virtual networks can be created using the following API. By changing the HTTP verb from POST to GET, existing virtual networks are read, and they can be updated (using PUT) or deleted (using DELETE).


```
POST http://10.84.14.2:8082/virtual-networks
Content-Type: application/json;
charset=UTF-8;
{"virtual-network":
                        {"fq_name": ["default-domain", "admin", "vn-blue"],
                        "network_ipam_refs":
                            [{"attr":
                                  {"ipam_subnets":
                                      [{"subnet":
                                          {"ip_prefix": "10.1.1.0", "ip_prefix_len":
24}
                                      }]
                                  },
                                   "to":
                                      ["default-domain", "admin", "default-network-
ipam"]
                             }]
                        }
}
```

- Network policies—These define how the instances are connected to the virtual networks and how the traffic flows.

```
POST http://10.84.14.2:8082/network-policys
Content-Type: application/json;
charset=UTF
{"network-policy":
                       {"fq_name": ["default-domain", "admin", "policy-red-blue"],
                           "network_policy_entries":
                               {"policy_rule":
                                    [{"direction": "<>", "protocol": "tcp",
                                    "dst_addresses":
                                        [{"virtual_network": "default-
domain:admin:vn-blue"}], "dst_ports":
 [{"start_port": 80, "end_port": 80}],
                                        "simple_action": "pass",
                                    "src_addresses":
                                        [{"virtual_network": "default-
domain:admin:vn-red"}], "src_ports":
[{"end_port": -1, "start_port": -1}]
                                          }]
                                }
                       }
}
```

- Service templates—These provide another abstraction layer that allows you to specify what parameters are needed to create a new service. Today, a service template can contain up to three virtual networks that specify how the traffic to and from the instance flows. Service templates are defined as issvcActivation/config/service.conf.

Example

New virtual security services need to be offered:

- Basic firewall, elite firewall, intrusion detection system (IDS), and intrusion prevention system (IPS)

```
[DEFAULTS]
                    api_server_ip=10.84.13.31
                    api_server_port=8082
                    proj_name=some_project_name
                    mgmt_vn=mgmt-virtual-network
                    left_vn=left-virtual-network
                    right_vn=right-virtual-network
                    max_instances=1
                    auto_scale=true
                    instance_name=some_instance_name
                    template_name=some_template_name
                    policy_name=some_policy_name
```

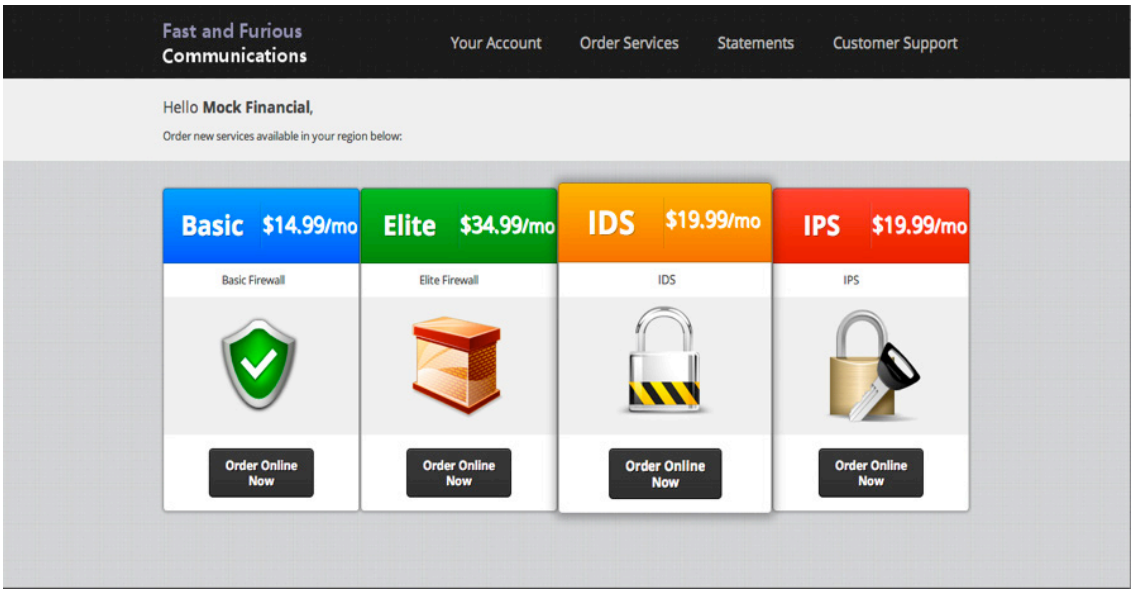

Figure 2: Customer Portal Example.

Images with a preconfigured security appliance for the aforementioned services are created and uploaded into an image folder of the service orchestration system.

From a network topology view, each customer is connected via an access network to the data center. In the data center, the security appliance is instantiated each time a subscriber establishes connection to the service provider network.

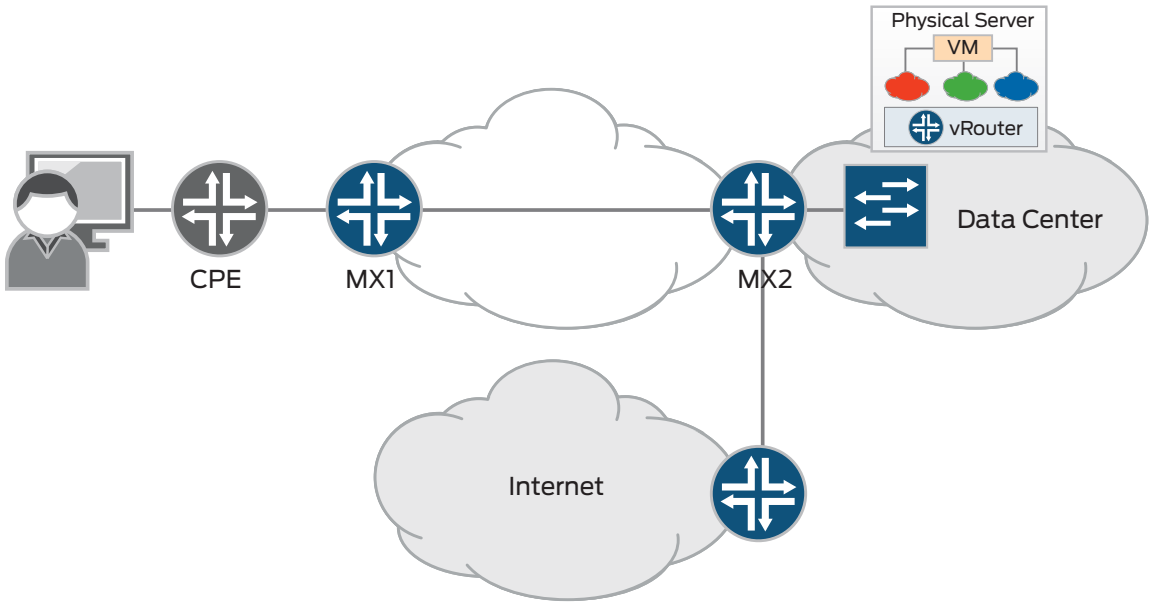

Figure 3: Virtual CPE network topology deployment example

The VM connects the red network to the customer CPE and the blue network to the Internet. The green network is for management of the security appliance.

Using REST API with method POST, new virtual networks are created. The following is an example of how to create vn-blue.

```
POST http://10.84.14.2:8082/virtual-networks
Content-Type: application/json;
charset=UTF-8;
{"virtual-network":
                    {
                    "fq_name":
                       ["default-domain", "admin", "vn-blue"],
                       "network_ipam_refs":
                          [{"attr":
                               {"ipam_subnets":
                                  [{"subnet":
                                       {"ip_prefix": "10.1.1.0", "ip_prefix_len":
24}
                                  }]
                                },
                    "to":
                       ["default-domain", "admin", "default-network-ipam"]
                          }]
                    }
}
```

Response

```
{"virtual-network":
                    {"href": "http://10.84.14.2:8082/virtual-network/c07f0ecf-
bfb6-4fbe-8f01-45580027d25b", "fq_name":
                          ["default-domain", "admin", "vn-blue"],
                          "name": "vn-blue",
                          "uuid": "c07f0ecf-bfb6-4fbe-8f01-45580027d25b"
                    }
}
```

We want to create a network policy that defines how the traffic is flowing to and from the instance. In the previous example, the bidirectional HTTP traffic is specified.

```
POST http://10.84.14.2:8082/network-policys
Content-Type: application/json;
charset=UTF;
                    {"network-policy":
                        {"fq_name": ["default-domain", "admin", "policy-red-
blue"],
                              "network_policy_entries":
                                {"policy_rule":
                                      [{"direction": "<>", "protocol": "tcp",
                                      "dst_addresses":
                                          [{"virtual_network": "default-
domain:admin:vn-blue"}], "dst_ports": [{"start_port": 80, "end_port": 80}],
                                                  "simple_action": "pass",
                                      "src_addresses":
                                          [{"virtual_network": "default-
domain:admin:vn-red"}], "src_ports": [{"end_port": -1, "start_port": -1}]
                                       }]
                                  }
                        }
                    }
Response
{"network-policy":
                    {"href": "http://10.84.14.2:8082/network-policy/507ce7ff-e277-
4508-937b-b18c6fd087e9",
                         "fq_name": ["default-domain", "admin", "policy-red-
blue"],
                         "name": "policy-red-blue",
                         "uuid": "507ce7ff-e277-4508-937b-b18c6fd087e9"
                    }
}
```

Once the needed virtual networks and network policies are created, the service can be instantiated, either by the end user via a GUI or another application via an API (as in the following example).

```
#create service instance
  def create_si(self):
#get service template
    try:
      st_obj = self._vnc_lib.service_template_read(fq_name=self._st_fq_name)
      st_prop = st_obj.get_service_template_properties()
      if st_prop == None:
   print "Error: Service template %s properties not found" %     (self._args.
template_name)
         return
     except NoIdError:
       print "Error: Service template %s not found" % (self._args.template_name)
       return
```

```
#check if passed VNs exist
    if self._args.left_vn:
       try:
         self._vnc_lib.virtual_network_read(fq_name = self._left_vn_fq_name)
       except NoIdError:
         print "Error: Left VN %s not found" % (self._left_vn_fq_name)
         return
    if self._args.right_vn:
       try:
         self._vnc_lib.virtual_network_read(fq_name = self._right_vn_fq_name)
       except NoIdError:
          print "Error: Right VN %s not found" % (self._right_vn_fq_name)
          return
#create si
   print "Creating service instance %s" % (self._args.instance_name)
   project = self._vnc_lib.project_read(fq_name=self._proj_fq_name)
   try:
     si_obj = self._vnc_lib.service_instance_read(fq_name=self._si_fq_name)
     si_uuid = si_obj.uuid
   except NoIdError:
     si_obj = ServiceInstance(self._args.instance_name, parent_obj = project)
     si_uuid = self._vnc_lib.service_instance_create(si_obj)
   si_prop = ServiceInstanceType(left_virtual_network = self._args.left_vn,
                    management_virtual_network = self._args.mgmt_vn,
                    right_virtual_network = self._args.right_vn)
#set scale out
    scale_out = ServiceScaleOutType(max_instances = self._args.max_instances,
                      auto_scale = self._args.auto_scale)
    si_prop.set_scale_out(scale_out)
    si_obj.set_service_instance_properties(si_prop)
    st_obj = self._vnc_lib.service_template_read(id = st_obj.uuid)
    si_obj.set_service_template(st_obj)
    self._vnc_lib.service_instance_update(si_obj)
 return si_uuid
#end create_si
```

Once the service is activated, operational information can be retrieved using the ops server. The data provided from the ops server can be used as a trigger for another service enablement. For example, the customer might be exceeding the available resources of the service instance and based on the subscriber policy, the ops server is allowed to automatically extend the service capacity and charge for it.

Conclusion

Designed to increase business innovation, Contrail APIs abstract the network by interoperating with open cloud orchestration systems. Contrail APIs ensure that applications are more robust and enable businesses to create customized management solutions for their customers' specific needs

Contrail APIs enable new network-aware applications to be created that provide control over networks. They create a programmable network that enables you to leverage the connections and intelligence embedded in the network to create customized management solutions for your company and your customers' specific needs. You can make better use of resources to create and use new insertion points for creating revenue streams. Contrail APIs make it simple to safely extract data from your network to use in applications. This data allows for the creation of solutions that can respond and react to your network, making applications more intelligent and providing better end-user experiences.

About Juniper Networks

Juniper Networks is in the business of network innovation. From devices to data centers, from consumers to cloud providers, Juniper Networks delivers the software, silicon and systems that transform the experience and economics of networking. The company serves customers and partners worldwide. Additional information can be found at www.juniper.net.

Corporate and Sales Headquarters
Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, CA 94089 USA
Phone: 888.JUNIPER (888.586.4737)
or +1.408.745.2000
Fax: +1.408.745.2100
www.juniper.net

APAC and EMEA Headquarters
Juniper Networks International B.V.
Boeing Avenue 240
1119 PZ Schiphol-Rijk
Amsterdam, The Netherlands
Phone: +31.0.207.125.700
Fax: +31.0.207.125.701

2000527-001-EN Sept 2015