

INCREASING NETWORK AVAILABILITY WITH AUTOMATED SCRIPTING

How to Use JUNOScript Automation to Avert Errors, Monitor Networks,
and Resolve Problems

Table of Contents

Executive Summary	1
Introduction	1
Introduction to JUNOScript Automation	2
Human Factors and JUNOS Software	2
Custom Procedures for Each Network	3
Using SLAX for Automated Scripting	4
Commit Scripts	4
How Commit Scripts Work	5
Commit Script Operational Flow	6
Commit Script Examples	7
Advantages of Commit Scripts	12
Operation Scripts	12
How Operation Scripts Work	13
Op Script Examples	14
1. Restarting an FPC	14
2. Customizing Show Interfaces Terse Output	15
Event Policy	16
How Event Policy Works	16
Event Policy Examples	17
1. Correlating Events Based on Receipt of Other Events Within a Specified Time Interval	17
2. Ignoring Events Based on Receipt of Other Events	17
3. Correlating Events Based on Event Attributes	18
4. Generating Internal Events	19
5. Dampening an Event	19
6. Controlling Event Policy Using a Regular Expression	20
7. Raising SNMP Traps	20
Using Scripts for Capacity Planning and Monitoring in a Router	21
Capacity Planning and Monitoring Examples	21
1. Setting and Monitoring an MIB Variable via RMON	21
2. Populating Custom Data into an Accounting File for Periodic Transfer and Processing	23
Scripting Best Practices	24
Conclusion	24
References	24
Appendix	25
Commit Script Boilerplate	25
Commit Script Examples	25
1. Imposing a Minimum MTU Setting	25
2. Controlling LDP Configuration	27
3. Creating a Complex Configuration Based on a Simple Interface Configuration	28
4. Controlling a Dual Routing Engine Configuration	31

Operation Script Examples	33
1. Restarting an FPC	33
2. Customizing Show Interfaces Terse Output	34
Capacity Planning and Monitoring Examples	36
1. Setting and Monitoring a MIB Variable via RMON	36
2. Populating Custom Data into an Accounting File for Periodic Transfer and Processing	39
About Juniper Networks	42

Table of Figures

Figure 1: Sources of network device downtime	1
Figure 2: SLAX and the XML output process	4
Figure 3: Commit script checkout process	6
Figure 4: Eventd process interaction	16

Executive Summary

Network outages can result from various types of issues and events, but for many organizations the most common cause is human error—a factor that may be overlooked while attention focuses on hardware, link, or software failures. With many potential sources of downtime, operations teams need ways to improve response times to network problems and, better yet, they need mechanisms that can address the underlying causes to proactively avert issues from happening in the first place.

Juniper Networks® JUNOS® Software, a single network operating system integrating routing, switching, security, and network services, offers advanced features and functionality to improve the availability and delivery of services, including tools that consider the human factors contributing to downtime. Among recent JUNOS innovations for more continuous systems are flexible scripting technologies that run on-box to avert configuration errors and accelerate problem identification and resolution.

JUNOScript Automation brings automated operations into today's networks through intelligent and customizable scripts. With these tools, teams can extend the expertise of their best engineers across the network. These scripts can prevent common operations mistakes by simplifying and validating configurations and also automating event detection and diagnostic procedures. Scripting enables a continuous improvement capability as engineers diagnose problems and then script proactive avoidance steps.

The introductory sections of this paper provide an overview of the advances that JUNOScript Automation brings to enterprise and service provider networks, and is targeted for network decision makers. The remainder of the paper provides detailed instructions for engineers implementing scripts.

Introduction

Leaders in operational excellence understand the necessity of building reliability into networks, from ensuring redundant connections to deploying equipment with a high mean time between failures. Yet, some may be surprised to learn that in many networks a majority of the outages result from human error. Figure 1 summarizes three primary causes of network downtime related to device outages, with relative scale gathered from various industry reports and customer perspectives.

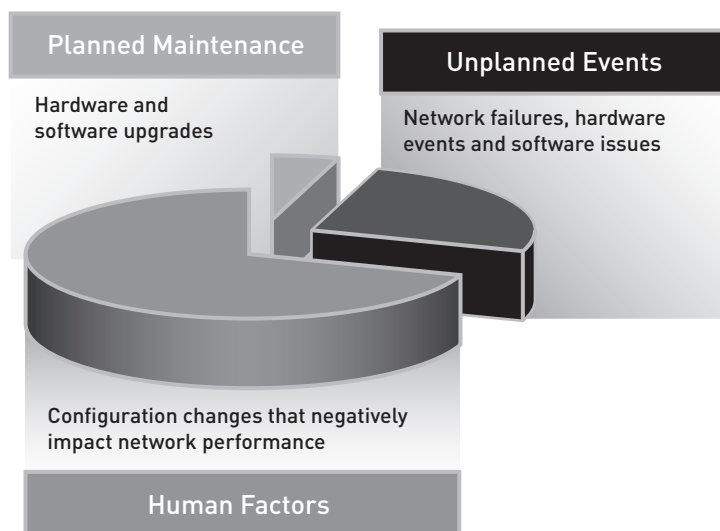


Figure 1: Sources of network device downtime

Most events related to human errors occur during the process of configuring routers or other network devices. Even the most experienced engineer knows it's more than possible to mistype, make a syntax error, or configure a new service in a way that brings unexpected (and undesirable) results: a firewall implemented on the wrong interface, a service link to a major customer broken, the wrong IP address on a filter list.

Further, as more and more services are enabled over IP networks, device configurations are becoming increasingly complex. As complexity increases, so too does the likelihood of an incorrect command. At the same time, an outage in a modern multiservice network carrying data, voice, and video can be extraordinarily expensive in terms of service-level agreement (SLA) penalties and damaged customer confidence.

With service continuity at risk, businesses need ways to improve their response times to network problems and, better yet, they need tools that will help them prevent these problems from happening in the first place. Operators need access to automated triage detection and diagnosis by the network device itself so that when an alert or page is received, the information needed to solve the problem is immediately at hand. And to be cost-effective, any solution needs to scale across an entire infrastructure.

Introduction to JUNOScript Automation

JUNOScript Automation offers an innovative solution to these challenges: a suite of powerful new tools that can validate candidate configurations, provide early detection of potential issues, and automate elements of problem resolution. With JUNOScript Automation, enterprises and service providers can prevent common operations mistakes and speed early response to issues.

Three types of scripts enforce compliance with the standards, policies, and requirements of your network. Commit scripts parse candidate configurations upon commit, expanding condensed configurations through macros, generating warnings, modifying the configuration, or even stopping a problematic candidate from becoming the active configuration. Operation (op) scripts and event policies enable customized network troubleshooting along with automated diagnosis and remediation. For example, a script could periodically check network warning parameters, such as high CPU usage, to provide proactive notification of leading indicators and accelerate problem identification and resolution.

These tools and features that directly address the impact of human factors on network availability complement a long list of JUNOS attributes for high system stability: a modular software design, disciplined processes of software development, error-resilient configuration, unified in-service software upgrade (ISSU), among others.

Operations teams will find that JUNOScript Automation provides an excellent complement to existing network automation systems. The existing systems offer substantial benefits for change management, provisioning, and monitoring, but their usefulness is limited when it comes to detecting and diagnosing network problems. Operations teams need to be able to identify potential issues in real time so that they can take immediate steps towards preventive action. Automated change management systems can only identify problems after the fact, as these packages collect information about system conditions reactively, by polling the device at predefined intervals. JUNOScript Automation is unique in that it provides immediate, on-box problem detection and resolution. The new scripts are always running, always alert to potential issues, and always ready to initiate repair.

Advanced Insight Solutions

Building on the pioneering capabilities first of internal scripts and then of JUNOScript Automation, Advanced Insight Solutions (AIS) introduces intelligent self-analysis capabilities directly into platforms run by JUNOS Software. AIS provides a comprehensive set of tools and technologies designed to enable Juniper Networks Technical Services with the automated delivery of tailored, proactive network intelligence and support services. By integrating advanced support intelligence into networking platforms, automating support steps, and providing proactive insight into JUNOS operations, AIS increases network availability and lowers operations costs.

Human Factors and JUNOS Software

Each operations organization develops its own set of network and security best practices, rules, and policies, ensuring that common standards are in place across the entire business infrastructure. For a new employee, no matter how thorough the training or extensive the previous experience, it takes time to get up to speed with new operational procedures and policies. Further, knowing what operational steps work best in finding and remedying problems is a lifelong learning experience.

Juniper engineers have long recognized the human factor of network outages. JUNOS includes multiple features for avoiding and recovering from operational mistakes during configuration changes. The JUNOS commit model for configuration provides straightforward version control, inline warning messages, and the ability to deactivate problematic configurations.

Operators update devices via a candidate configuration that is a copy of the running configuration. For the configuration to become active, these users must explicitly commit their changes after entering and reviewing all modifications, and the system automatically checks for syntax or other incorrect configuration constructions. If one

or more errors are found, the built-in scripts prevent the candidate configuration from becoming active and inform the user of the discovered errors. These safeguards were introduced with the first version of JUNOS Software and our engineers have regularly enhanced the scripts ever since, based on any invalid command combinations they can anticipate.

Custom Procedures for Each Network

Every network, however, is unique. While Juniper engineers can anticipate invalid command combinations, they cannot create scripts that prevent violations of configuration standards on a particular network. In addition, what is acceptable in one network may be unacceptable in another. Each customer organization should have the ability to ensure that their own standards are followed and their device configurations are consistent. It is JUNOScript Automation that makes this possible.

With JUNOScript Automation, users can create scripts that reflect their own business needs and procedures. Most often, the scripts are written by an organization's most experienced engineers—those who can flag potential errors in basic configuration elements such as interfaces, peering, and VPNs. The scripts automate network troubleshooting and quickly detect, diagnose, and fix problems as they occur. In this way, new personnel running the scripts benefit from their predecessors' long-term knowledge and expertise. Networks using JUNOScript Automation increase productivity, reduce OPEX, and surpass conventional standards for high availability (HA).

JUNOScript Automation offers three powerful scripting tools:

- Commit scripts enable users to create custom scripts that ensure configurations are in compliance with each network's rules and standards. When a noncompliant configuration is discovered, a commit script can produce a warning message, prevent the configuration from becoming active, or even correct the configuration on the fly. The true power of commit scripts becomes evident with their macro capabilities: users can create a macro that takes simple configuration variables as input and outputs a complete configuration, guaranteeing full consistency among all device configurations. For example, macros represent an ideal way to standardize a northbound interface with Network Management and Operation Support System applications.
- Operation (op) scripts allow users to create scripts that automate network troubleshooting and management. An op script can monitor the device and take specified actions ranging from simple notifications to automatic diagnosis and correction of detected problems. With op scripts, an organization can extend the expertise of its most experienced engineers to all operations personnel by creating step-by-step diagnostic procedures to guide staff toward fast, accurate problem resolution.
- Event policies define policies that take a specified action when a particular event notification message is received from one of the JUNOS Software modules.

Op scripts and event policies work together to automate early warning systems that not only detect emerging problems, but can also take immediate steps to avert further outages and restore normal operations. Customers can customize the scripts for network monitoring and data correlation, change detection, automated troubleshooting sequences and remediation guidance, or even fixes specific to their network. By watching for network events specified by each organization—warning parameters that signal potential problems such as a line card crash, routing failure, or memory overload—operations teams can respond more quickly with corrective actions.

For example, a typical op script might initiate a sequence such as: "If CPU usage spikes above 75 percent, disable services 1, 2, and 3 to prevent failures," or "Make sure that each ATM interface has no more than 1,000 PVCs connected." In this way, teams can better control the series of operations events from the first leading indicator. Scripts enhance operations expertise and conserve valuable time after events occur. By capturing more directly relevant information faster, scripts give operations teams more options for reacting to minor issues rather than letting unchecked, escalating events lead to worse-case scenarios.

A key benefit of operation scripts is their ability to iteratively narrow down the cause of network problems. Even if an op script doesn't uncover the root cause of a problem, the scripts can give operators a running start that can be immensely valuable. Rapid problem diagnosis is crucial during an outage; it is not uncommon for an operations team to spend hours diagnosing a problem that ultimately takes only a few minutes to repair.

Using SLAX for Automated Scripting

Juniper engineers have developed a new scripting language that expedites the process of creating commit scripts and op scripts for customer networks. SLAX, or Stylesheet Language Alternative syntax, will be immediately familiar to users of Perl, as SLAX uses a Perl-like syntax and the same straightforward language constructs.

XSLT, or Extensible Stylesheet Language Transformations, is another option for writing scripts but XSLT is considered verbose and difficult to read and write. Both SLAX and XSLT were designed for use with XML, which encapsulates all commands entered on Juniper devices. SLAX is best seen as an addition to XSLT, as the underlying SLAX constructs are completely native to XSLT.

Figure 2 shows how SLAX works with XSLT and the JUNOS management process (mgd) to format XML output. Before JUNOS invokes the XSLT processor, the software converts SLAX constructs such as if/then/else to equivalent XSLT constructs such as <xsl:choose> and <xsl:if>, and builds an XML tree identical to the one produced when the XML parser reads an XSLT document.

SLAX is used in this paper's examples, and we recommend its use for customers writing scripts.

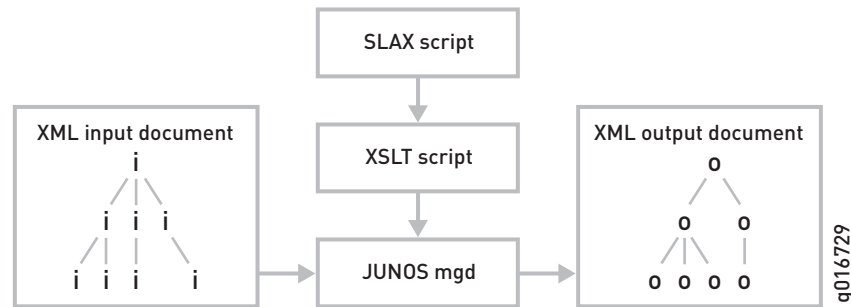


Figure 2: SLAX and the XML output process

Commit Scripts

JUNOScript Automation commit scripts provide a way to enforce each organization's custom configuration rules. Each time an engineer commits a new candidate configuration, the commit scripts run and inspect that configuration. If the configuration violates the network's operational rules, the script can instruct JUNOS to perform various actions. For example, the script can:

- Generate custom error messages
- Generate custom warning messages
- Generate custom system log (syslog) messages
- Change the configuration

Users can also write custom macros, creating syntax that simplifies the task of configuring a device. By itself, the customized syntax has no operational impact on the device. A corresponding commit script macro uses the custom syntax as input and generates standard JUNOS configuration statements for setting up the desired functionality.

Consider a network design that requires every core-facing interface enabling ISO protocols to also enable MPLS. At commit time, a commit script inspects the configuration and issues an error if this requirement is not met. The error causes the commit operation to fail and forces the user to update the configuration to comply. Or, instead of an error, the commit script can issue a warning about the configuration problem and then automatically correct it by changing the configuration to enable MPLS on all interfaces. A system log message can also be generated, indicating that corrective action was taken.

A commit script could also be used to enforce physical card placement and wiring standards, as well as logical configuration. This would ensure that only designated ports are configured for customer connections, while other ports are reserved for connectivity to the core of the network.

Another elegant way to enable the two protocols would be to define a commit script macro that enables ISO protocols and MPLS when the macro is applied to an interface. Configuring this macro simplifies the configuration task while ensuring that both protocols are configured together.

How Commit Scripts Work

Users enable a commit script by listing the names of one or more commit script files at the `[edit system scripts commit]` hierarchy level. Commit scripts are invoked during the commit process before the standard JUNOS validity checks.

When the user performs a commit operation, JUNOS executes each script in turn, passing the candidate configuration through the defined commit scripts. The scripts inspect the candidate, perform the necessary tests and validations, and generate a set of instructions, requesting the software to perform certain actions. These actions include generating error, warning, and system log messages. If errors are generated, the commit operation fails and the candidate configuration remains unchanged. This is also the behavior that occurs with standard validity check errors in JUNOS.

If a commit script changes the system configuration, the changed configuration is loaded and standard validation checks are performed. If the candidate configuration passes all standard validation checks, it becomes the active, operational device configuration.

How to Enable a Commit Script

1. Write a commit script based on the examples in this paper.
2. Copy the script to the `/var/db/scripts/commit` directory. Only users in the superuser JUNOS login class can access and edit files in this directory.

To enable the script on a platform with dual Routing Engines, copy the script to the `/var/db/scripts/commit` directory on each Routing Engine. The `commit synchronize` command does not automatically copy the scripts from one RE directory into the other RE directory.

3. Enable the script by including a file statement at the `[edit system scripts commit]` hierarchy level. Only users in the superuser class can configure commit scripts.

```
[edit system scripts commit]
file filename;
```

4. Issue a `commit` command.

JUNOS provides several tools to manage and monitor commit script operation, including source control, traceoptions, and monitoring commands. For more information, see the "Configuring and Troubleshooting Commit Scripts" section in the *JUNOS Configuration and Diagnostic Automation Guide*.

Commit Script Operational Flow

Figure 3 shows the flow of operations when commit scripts are added to the standard JUNOS commit model.

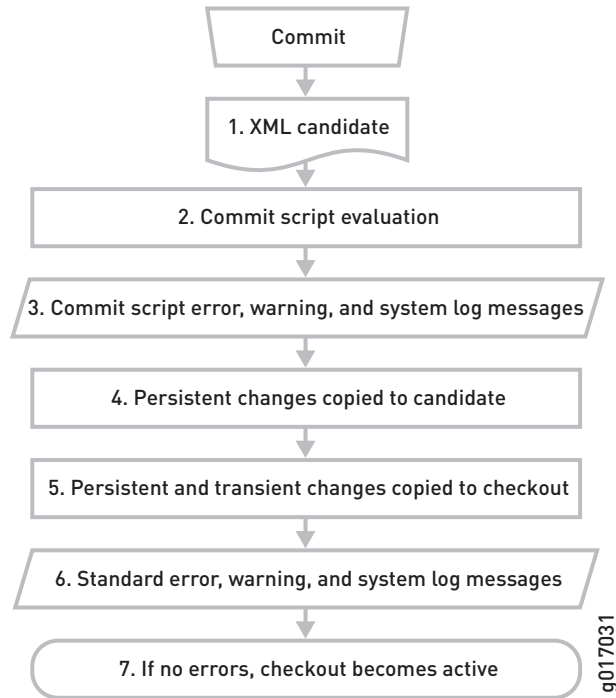


Figure 3: Commit script checkout process

Steps 2 and 3 ensure that the candidate configuration conforms to each network’s custom rules through the commit scripts. Persistent changes are then added to the candidate configuration in Step 4. Step 6 is the standard JUNOS validity checks.

Useful Abbreviations

API Application Programming Interface	OID Object Identifier
CLI Command Line Interface	OSS Operations Support System
FPC Flexible PIC Concentrator	PFE Packet Forwarding Engine
IGP Interior Gateway Protocol	PIC Physical Interface Card
IPv4 IP Version 4	RE Routing Engine
LDP Label Distribution Protocol	RMON Remote MONitoring
mgd JUNOS management process	SLA Service Level Agreement
MIB Management Information Base	SLAX Stylesheet Language Alternative syntaX
MPLS MultiProtocol Label Switching	SNMP Simple Network Management Protocol
MTU Maximum Transmission Unit	XML Extensible Markup Language
NMS Network Management System	XSLT Extensible Stylesheet Language Transformations

A second test ensures that all LDP-enabled interfaces are configured for an interior gateway protocol (IGP). As with LDP, some interfaces can be exempted from the test by including the `apply-macro no-igp` statement at the `[edit protocols ldp interface interface-name]` hierarchy level. If that statement is not included and no IGP is configured, a warning is emitted.

```
.....  
match configuration {  
[. . .]  
    if ($ldp) {  
        for-each ($isis/interface/name | $ospf/area/interface/name) {  
[. . .]  
            if (not(..apply-macro[name == "no-ldp"]) &&  
                not($ldp/interface[name == $ifname])) {  
                // Print Warning that ldp is not enabled for this interface";  
            }  
        }  
    }  
  
    for-each (protocols/ldp/interface/name) {  
[. . .]  
  
        if (not(apply-macro[name == "no-igp"]) &&  
            not($isis/interface[name == $ifname]) &&  
            not($ospf/area/interface[name == $ifname])) {  
            // Print Warning that ldp-enabled interface does not  
            // have an IGP configured  
        }  
    }  
}  
}
```

.....

3. Controlling a Dual Routing Engine Configuration

JUNOS configurations may be complex if the routing platform has redundant (dual) Routing Engines (REs). The example below shows how commit scripts can be used to simplify and control the configuration of dual RE platforms.

JUNOS Software supports two special configuration groups: `re0` and `re1`. When these groups are applied using an `apply-groups [re0 re1]` statement, they take effect if the Routing Engine name matches the group name. Statements included at the `[edit groups re0]` hierarchy level are inherited only on the Routing Engine named RE0, and statements included at the `[edit groups re1]` hierarchy level are inherited only on the Routing Engine named RE1.

The example below shows two commit scripts. The first script, `ex-dual-re.slax`, emits a warning if the system host-name statement, any IPv4 interface address, or the `fxp0` interface configurations are not part of the `[re0 re1]` configuration groups.

The second script, `ex-dual-re2.slax`, determines whether the hostname is configured and whether it is configured in a configuration group. If the hostname is not configured and whether it is configured in a configuration group. If the hostname is not configured at all, an error message is emitted. The first `if` conditional check ensures that the script does nothing if the hostname is already configured in a configuration group. The second `else if` construct takes effect when the hostname is configured in the target configuration. In this case, the script generates a transient change that places the hostname configuration into the `re0` and `re1` configuration groups, copies the configured hostname into those groups, concatenates each group hostname with `-RE0` and `-RE1`, and deactivates the hostname in the target configuration so that the configuration group hostnames can be inherited.

```

.....
Script 1: ex-dual-re.slax
match configuration {
  for-each (system/host-name |
    interfaces/interface/unit/family/inet/address |
    interfaces/interface[name = 'fxp0']) {
    if (not(@junos:group) or not(starts-with(@junos:group, 're'))) {
      // Print Warning - statement should not be in target
      configuration on dual RE system
    }
  }
}
Script 2: ex-dual-re2.slax
match configuration {
  var $hn = system/host-name;
  if ($hn/@junos:group) {
  }
  else if ($hn) {
    <transient-change> {
      <groups> {
        <name> "re0";
        <system> {
          <host-name> $hn _ '-RE0';
        }
      }
      <groups> {
        <name> "re1";
        <system> {
          <host-name> $hn _ '-RE1';
        }
      }
      <system> {
        <host-name inactive="inactive">;
      }
    }
  }
  else {
    <xnm:error> {
      <message> "Missing [system host-name]";
    }
  }
}
}
.....

```

4. Creating a Complex Configuration Based on a Simple Interface Configuration

The example below uses a commit script macro to automatically expand a simple interface configuration. The script generates a transient change that assigns a default encapsulation type, configures multiple routing protocols on the interface, and applies multiple configuration groups.

In the example, the JUNOS Software management process (mgd) inspects the configuration to look for `apply-macro params` statements at the `[edit interfaces interface-name]` hierarchy level. When the script finds an `apply-macro params` statement, it does the following:

- Applies the `interface-details` configuration group to the interface.
- Includes the value of the `description` parameter at the `[edit interfaces interface-name description]` hierarchy level.
- Includes the value of the `encapsulation` parameter at the `[edit interfaces interface-name encapsulation]` hierarchy level. If the `encapsulation` parameter is not included in the `apply-macro params` statement, the script sets the encapsulation to `cisco-hdlc` by default.
- Sets the logical unit number to 0 and determines whether the `inet-address` parameter is included in the `apply-macro params` statement. If it is, the script includes the value of the `inet-address` parameter at the `[edit interfaces interface-name unit 0 family inet address]` hierarchy level.
- Includes the interface name at the `[edit protocols rsvp interface]` hierarchy level.
- Includes the `level 1 enable` and `metric` statements at the `[edit protocols isis interface interface-name]` hierarchy level.
- Includes the `level 2 enable` and `metric` statements at the `[edit protocols isis interface interface-name]` hierarchy level.
- Determines whether the `isis-level-1` or `isis-level-1-metric` parameter is included in the `apply-macro params` statement. If one or both of these parameters is included, the script includes the `level 1` statement at the `[edit protocols isis interface interface-name]` hierarchy level. If the `isis-level-1` parameter is included, the script also includes the value of the `isis-level-1` parameter (`enable` or `disable`) at the `[edit protocols isis interface interface-name level 1]` hierarchy level. If the `isis-level-1-metric` parameter is included, the script includes the value of the `isis-level-1-metric` parameter at the `[edit protocols isis interface interface-name level 1 metric]` hierarchy level.

```

.....
match configuration {
    var $top = .;

    for-each (interfaces/interface/apply-macro[name == "params"]) {
        var $description = data[name == "description"]/value;
        var $inet-address = data[name == "inet-address"]/value;
        var $encapsulation = data[name == "encapsulation"]/value;
        var $clocking = data[name == "clocking"]/value;
        var $isis-level-1 = data[name == "isis-level-1"]/value;
        var $isis-level-1-metric = data[name == "isis-level-1-
metric"]/value;
        var $isis-level-2 = data[name == "isis-level-2"]/value;
        var $isis-level-2-metric = data[name == "isis-level-2-
metric"]/value;
        var $ifname = ../name _ ".0";
        <transient-change> {
            <interfaces> {
                <interface> {
                    <name> ../name;
                    <apply-groups> {
                        <name> "interface-details";
                    }
                    if ($description) {
                        <description> $description;
                    }
                }
            }
        }
    }
}

```

```

    <encapsulation> {
        if (string-length($encapsulation) > 0) {
            expr $encapsulation;

        } else {
            expr "cisco-hdlc";
        }
    }
    <unit> {
        <name> "0";
        if (string-length($inet-address) > 0) {
            <family> {
                <inet> {
                    <address> $inet-address;
                }
            }
        }
    }
}
<protocols> {
    <rsvp> {
        <interface> {
            <name> $ifname;
        }
    }
    <isis> {
        <interface> {
            <name> $ifname;
            if ($isis-level-1 || $isis-level-1-metric) {
                <level> {
                    <name> "1";
                    if ($isis-level-1) {
                        <xsl:element name = $isis-level-1>;
                    }
                    if ($isis-level-1-metric) {
                        <metric> $isis-level-1-metric;
                    }
                }
            }
            if ($isis-level-2 || $isis-level-2-metric) {
                <level> {
                    <name> "2";
                    if ($isis-level-2) {
                        <xsl:element name = $isis-level-2>;
                    }
                    if ($isis-level-2-metric) {
                        <metric> $isis-level-2-metric;
                    }
                }
            }
        }
    }
}
<ldp> {
    <interface> {
        <name> $ifname;
    }
}
<class-of-service> {
    <interfaces> {

```

```
        <name> ../name;
        <apply-groups> {
            <name> "cos-details";
        }
    }
}
}
```

Advantages of Commit Scripts

Commit scripts provide operators with an operational safety net that is invaluable in today's complex networks. By creating a library of scripts that targets typical problems, users can enforce custom business policies and substantially reduce the likelihood of human error.

Here are some examples of everyday actions that operators can carry out with commit scripts:

- **Basic sanity test**—such as to ensure that the [edit interfaces] and [edit protocols] hierarchies have not been accidentally deleted.
- **Consistency check**— such as to ensure that every T1 interface configured at the [edit interfaces] hierarchy level is also configured at the [edit protocols rip] hierarchy level.
- **Dual RE configuration check**—such as to ensure that the re0 and re1 configuration groups are set up correctly. The inherited values in configuration groups can be inadvertently overridden in the target configuration. A commit script can ensure that nothing in the target configuration is blocking proper inheritance of configuration group settings.
- **Interface density**—such as to ensure that there are not too many channels configured on a channelized interface.
- **Link scaling**—such as to ensure that SONET/SDH interfaces never have an MTU size less than the specified size, such as 4 kilobytes.
- **Import policy check**—such as to ensure that an IGP does not use an import policy that accidentally imports the full routing table.
- **Cross-protocol checks**—such as to ensure that all LDP-enabled interfaces are configured for an IGP, or to ensure that all IGP-enabled interfaces are configured for LDP.
- **IGP design check**—such as to ensure that Level 1 IS-IS routers are never enabled.
- **Implementation consistency**—such as to ensure that all customer-facing interfaces are located on dedicated FPCs and PIC slots rather than core-facing FPCs. This design rule guards against customer outages during system maintenance.

Operation Scripts

The JUNOScript Automation operation scripts (op scripts) automate many common troubleshooting and network management tasks. An op script can automatically diagnose and fix device problems. Op scripts can run an operational mode command, process the output, determine the next appropriate action, and repeat the process until the source of the problem is determined and reported to the command-line interface (CLI). For instance, an operation script can be used to monitor the overall status of a device, such as periodically checking network warning parameters. If there is a known problem in the JUNOS Software, the script can ensure that the device is configured to avoid or work around the problem. If there is a periodic problem in the JUNOS Software, an event policy can be configured that detects the periodic error condition and executes an operation script to work around it (see the event policy section below). An op script can also change a configuration in response to a problem.

Commit scripts and operation scripts are essentially the same instrument, called at different times, with different input and different output. The differences between commit and op scripts are as follows:

- Input—commit scripts always receive the candidate configuration as input. Op scripts receive no automatic input.
- Execution—commit scripts are executed automatically at commit time only. Op scripts are executed any time, either manually or automatically in response to an event. For example, both commit and op scripts can inspect and change a configuration. With commit scripts, the inspection occurs each time a new candidate configuration is committed. With op scripts, the user decides to manually run the op script.

How to Enable an Operation Script

Follow these steps to use an op script:

1. Write an op script based on the examples in this paper.
2. Copy the script to the `/var/db/scripts/op` directory. Only users in the superuser JUNOS login class can access and edit files in this directory.

To enable the script on a platform with dual Routing Engines, copy the script to the `/var/db/scripts/op` directory on each Routing Engine. The `commit synchronize` command does not automatically copy the scripts from one RE directory into the other RE directory.

3. Enable the script by including a `file` statement at the `[edit system scripts op]` hierarchy level. Only users in the superuser class can configure op scripts.

```
[edit system scripts op]
```

```
file filename;
```

4. Issue a commit command.

For more information, see the “Summary of Op Script Configuration Statements” section in the JUNOS Configuration and Diagnostic Automation Guide.

How Operation Scripts Work

Op scripts are enabled by listing the names of one or more op script files at the `[edit system scripts op]` hierarchy level. The scripts are invoked from the command line or from within an event policy, and executed by issuing an `op filename` or `op filename-alias operational mode` command:

```
[system scripts op]
traceoptions {
    flag all;
}
file dead-peers.slax {
    description "Diagnose issues with dead peers";
    arguments {
        peer {
            description "Peer to diagnose";
        }
    }
}
file op-bchip.slax {
    description "B-Chip dump";
}
file op-host.slax {
    description "simple reachability tests";
}
```

```

user@host> op ?
Possible completions:
<script>          Name of script to run
dead-peers        Diagnose issues with dead peers
op-bchip          B-Chip dump
op-host           simple reachability tests

```

```

user@host> op dead-peers ?
Possible completions:
<[Enter]>         Execute this command
<name>           Argument name
detail           Display detailed output
peer             Peer to diagnose
|               Pipe through a command

```

```

user@host> op dead-peers peer 10.1.2.3

```

.....

An op script can also be executed automatically in response to an event notification (see the event policy section for more details).

Op Script Examples

The following examples use operation script pseudocode to illustrate basic scripts for common operational tasks. Complete algorithms for each script are provided in the Appendix.

1. Restarting an FPC

This example restarts a Flexible PIC Concentrator (FPC) and modifies the output of the `request chassis fpc` command slightly to include the number of the restarting FPC:

```

.....
var $arguments = {
  <argument> {
    <name> "slot";
    <description> "Slot number of the FPC";
  }
}
param $slot;

match / {
  <op-script-results> {

var $restart = {
  <command> 'request chassis fpc slot ` _ $slot _ ` restart';
}
var $result = jcs:invoke($restart);

<output> {
  // Print the output of the command among others
}
}
}
}
.....

```

2. Customizing Show Interfaces Terse Output

This example shows how to create an op script that performs the following steps:

- Customizes the output of the `show interfaces terse` command to display only information of interest.
- Uses command-line arguments to provide additional customization, such as interface and protocol-specific details.
- By default, output from the `show interfaces terse` command is structured as follows. User input is in bold:

```

.....
user@host> show interfaces terse
Interface          Admin Link Proto   Local          Remote
dsc                 up    up
fxp0                up    up
fxp0.0              up    up   inet    192.168.71.246/21
fxp1                up    up
fxp1.0              up    up   inet    10.0.0.4/8
.....

```

In the example below, op script `ex-interface` outputs information related to only the interface name command-line argument.

```

.....
user@host> op ex-interface interface fxp0
Interface          Admin Link Proto   Local          Remote
fxp0.0              This is the Ethernet Management interface.
                                inet    192.168.71.246/21
.....

```

In the next example, op script `ex-interface` outputs information related to only the protocol command-line argument.

```

.....
user@host> op ex-interface protocol inet
Interface          Admin Link Proto   Local          Remote
fxp0.0              This is the Ethernet Management interface.
                                inet    192.168.71.246/21
fxp1.0              inet    10.0.0.4/8
lo0.0               inet    127.0.0.1      --> 0/0
lo0.16385           inet
.....

```

See the Appendix for further details about customizing command output.

Event Policy

Event policy plays a critical role in JUNOScript Automation, streamlining complex operations and accelerating network troubleshooting. Event policies can automatically detect a specified network event and take the appropriate action—that is, the action previous experience has shown to work in a specific network environment. A typical event policy anticipates a scenario such as: “If interface X and VPN Y go down, execute op script XYZ and log a customized message.”

To create an event policy that diagnoses device faults or error conditions, relevant information is needed about the state of the platform. State information can be derived from *event notifications*, which will henceforth simply be called *events*. Events can originate as system log messages or SNMP traps. A JUNOS process called the *event process* (eventd) receives event notifications from other JUNOS processes, such as the routing protocol process (rpd) and the management process (mgd).

After the eventd process receives events, a set of *event policies* instructs eventd to select and correlate specific events. The event policies also perform a set of actions, which can include invoking a JUNOS command or invoking an op script, creating a log file containing the output of these commands, and uploading the file to a given destination.

Figure 4 shows how eventd interacts with other JUNOS Software processes.

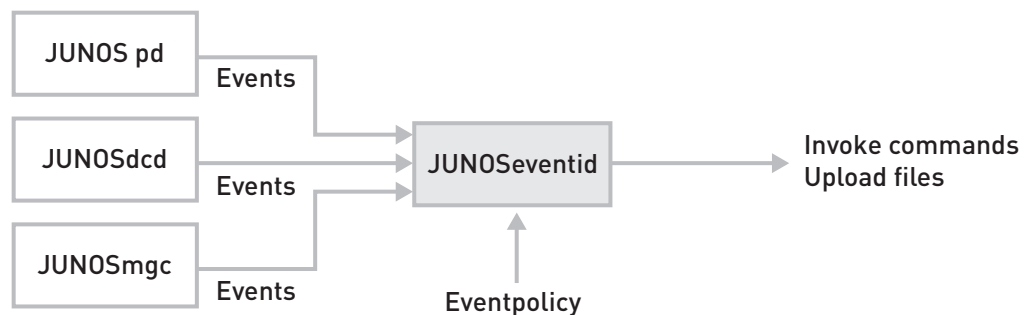


Figure 4: Eventd process interaction

An event policy is an if-then-else construct. Event policies define actions to be executed by the eventd process on receipt of an event. Multiple policies can be set up to process an event, and the policies will be executed in the order in which they appear in the configuration. Multiple actions can also be configured for each policy. These actions will also be executed in the order in which they appear in the configuration.

How Event Policy Works

To see a list of all events that can be referenced in an event policy, enter a help syslog ? command:

```

.....
user@host> help syslog ?
Possible completions:
<syslog-tag> System log tag
ACCT_ACCOUNTING_FERROR Error occurred during file processing
ACCT_ACCOUNTING_FOPEN_ERROR Open operation failed on file
[. . .]
.....

```

Based on the event policy, the eventd process can correlate two or more events and perform the following actions:

- Ignore the event—eventd does not generate a system log message for this event or process any further policy instructions for it.
- Upload a file to a specified destination—a transfer delay can be specified so that on receipt of an event, the file upload begins after the configured delay. For example, when uploading a core file, a transfer delay can ensure that the file has been completely generated before the upload begins.
- Execute JUNOS operational mode commands on receipt of an event—the XML or text command output is stored in a file, which will later be uploaded to a specified URL.

- Execute a JUNOS op script—variables can be included in the arguments section to pass to the op script, making it possible to incorporate data from the triggering event into the op script.
- Raise an SNMP trap.

Event Policy Examples

The following examples use pseudocode to illustrate event policies for common operational tasks. Complete algorithms for each policy are provided in the Appendix.

1. Correlating Events Based on Receipt of Other Events Within a Specified Time Interval

In the following policy, when the chassis process detaches the interface devices for a PIC within 80 seconds of a user-requested candidate configuration commit, the chassis hardware status and the active configuration are logged and saved to the output filename `my_cmd_out` specified in the policy.

Two archive sites are specified in this example. The device attempts to transfer to the first archive site in the list, moving to the next site only if the transfer fails.

```

.....
event-options {
  policy 1 {
    events [ CHASSISD_IFDEV_DETACH_PIC ];
    within 80 events [ UI_COMMIT ];
    then {
      execute-commands {
        commands {
          " show chassis hardware";
          " show configuration";
        }
        output-filename my_cmd_out;
        destination policy-1-command-dest;
      }
    }
    destinations {
      policy-1-command-dest {
        archive-sites {
          http://robot@my.big.com/a/b;
          http://robot@my.little.com/a/b;
        }
      }
    }
  }
}
.....

```

2. Ignoring Events Based on Receipt of Other Events

The scaling and performance characteristics of fault management network management systems (NMS) make it necessary for network operators to reduce the number of event flows from routers, switches, and security devices. Limiting the severity of errors that reach the fault collectors helps cut back on event flows, but as a consequence, important “early warning” events that could signal a major network outage may be ignored. Furthermore, during a serious network outage, uncorrelated events could flood the fault management system and adversely affect the entire fault management application.

JUNOS event policies help address this problem by suppressing unwanted events while reporting correlated events that are deemed important.

In the following policy, when the chassis process creates the initial interface device for a newly installed PIC or pseudodevice within 80 seconds of a user-requested candidate configuration commit, the event that triggered the policy is ignored, that is, system log messages are not created.

```
.....  
event-options {  
  policy 1 {  
    events [ CHASSISD_IFDEV_CREATE_NOTICE ];  
    within 80 events [ UI_COMMIT ];  
    then {  
      ignore;  
    }  
  }  
}
```

.....

3. Correlating Events Based on Event Attributes

The next example demonstrates another way to help external fault management applications cope with fault processing loads. In this example, a policy is configured that correlates two or more events. If the correlated events occur within 500 seconds of each other, they cause particular actions to be taken.

In the policy below, the two events are correlated only if two of their parameter values match. Matching on attributes of both events ensures that the two events are related. In this case, both the interface addresses and the physical interface (ifd) names must match.

The `RPD_RDISC_NOMULTI` error occurs when an interface is configured for router discovery, but the interface does not support IP multicast operations as required. The `RPD_KRT_IFDCHANGE` error occurs when rpd sends a request to the kernel to change the state of an interface, and the request fails. In the example, `RPD_RDISC_NOMULTI.interface-name` is designated `so-0/0/0.0`, and `RPD_KRT_IFDCHANGE.ifd-index` is designated `so-0/0/0`.

```
.....  
event-options {  
  policy 1 {  
    events rpd_rdisc_nomulti;  
    within 500 events rpd_krt_ifdchange;  
    attributes-match {  
      rpd_rdisc_nomulti.interface-address equals  
        rpd_krt_ifdchange.address;  
      rpd_rdisc_nomulti.interface-name starts-with  
        rpd_krt_ifdchange.ifd-index;  
    }  
  }  
}
```

.....

4. Generating Internal Events

Internal events are events the user creates to trigger execution of a policy. Internal events are not generated by JUNOS processes, and they have no associated system log messages. An internal event can be generated based on a time interval or the time of day.

In the following example, an internal event called `EVERY-ONE-HOUR` is generated every hour (3600 seconds). If 3601 seconds pass and the event has not been generated, appropriate actions are taken.

```
.....
event-options {
  generate-event every-one-hour time-interval 3600;
  policy check-heartbeat {
    events every-one-hour;
    within 3601 not events every-one-hour;
    then {
      ...
    }
  }
}
.....
```

5. Dampening an Event

Some low-severity events are generated very frequently, whereas more severe events can be generated repeatedly during a single outage and within a short period of time. In these situations, executing a policy multiple times for each instance of the event is resource intensive. Event dampening makes it possible to optimize and streamline the execution of policies by ignoring instances of events that arrive within a short time interval.

In the following example, an action is taken only if the eventd process has not received another instance of the event within the past 60 seconds. If an instance of the event has been received within the last 5 seconds, the policy is executed and a system log message for the event is not created again.

```
.....
event-options {
  policy dampen-policy {
    events event1;
    within 60 events event1;
    then {
      ignore;
    }
  }
  policy policy {
    events event1;
    then {
      [. . .]
    }
  }
}
.....
```

6. Controlling Event Policy Using a Regular Expression

The following policy is executed only if the `interface-name` attribute in both SNMP traps (`SNMP_TRAP_LINK_DOWN` and `SNMP_TRAP_LINK_UP`) match each other, and the `interface-name` attribute in the `SNMP_TRAP_LINK_DOWN` trap starts with the letter `t`. This means that the policy is executed only for T1 (`t1-`) and T3 (`t3-`) interfaces. The policy is not executed when the eventd process receives traps from other interfaces.

```

event-policy {
  policy pol6 {
    events snmp_trap_link_down;
    within 120 events snmp_trap_link_up;
    attributes-match {
      snmp_trap_link_up.interface-name equals
        snmp_trap_link_down.interface-name;
      snmp_trap_link_down.interface-name matches "^t";
    }
    then {
      execute-commands {
        commands {
          "show interfaces {${$.interface-name}";
          "show configuration interfaces {${$.interface-name}";
        }
        output-filename config.txt;
        destination bsd2;
        output-format text;
      }
    }
  }
}

```

7. Raising SNMP Traps

An event policy action can be configured that raises SNMP traps for events based on system log messages. This makes it possible for an SNMP trap-based application to notify a fault management NMS when an important system log message is generated. Any system log message for which there are no corresponding traps can be converted into a trap. This is valuable if NMS traps rather than system log messages are used to monitor the network.

The following example raises a trap and executes an associated event policy in response to an event:

```

event-options {
  policy p1 {
    events ui_mgd_terminate;
    then {
      raise-trap;
      event-script bgp.slax {
        arguments {
          destination {${ui_mgd_terminate.destination};
          code 2;
        }
        output-filename bgp-out;
        destination bsd3;
      }
    }
  }
}

```

Using Scripts for Capacity Planning and Monitoring in a Router

As the number of routers managed by a typical NMS grows and the complexity of the routers themselves increases, it becomes increasingly impractical to define a single set of metrics to determine whether or not a router is operating within its capacity limits, much less monitor these metrics from a central NMS application. Monitoring from a central NMS also places high computational demands on the router.

Because each of Juniper's customers uses a different set of applications, network types, and service offerings, the set of capacity metrics that customers need to monitor is unique as well. Rather than try to design a common capacity monitoring feature that addresses all operational needs, Juniper Networks gives customers and support staff the tools to customize their own applications and leverage the infrastructure already available on the routers. Further, by taking advantage of the router's ability to monitor itself and notify an NMS application on an exception basis, it is possible to address the scaling issues that arise as customer networks expand.

Some core capacity issues are addressed by the current JUNOS syslog/trap mechanisms, which generate notifications when any part of the system fails. Other existing features (health monitor, Remote Monitoring RMON alarms, and so on) can be used to generate early warnings—before a failure occurs—for a core set of metrics that is common to all router applications.

Beginning with JUNOS 8.4, op scripts have been enhanced as follows:

- The API to the SNMP MIB infrastructure allows an op script to populate a set of MIB tables—defined specifically for this feature with any number of desired values. The contents of these tables are available via the usual external SNMP requests as well as internal (RMON alarm) mechanisms.
- The API to the accounting feature allows an op script to populate any arbitrary accounting files with any number of desired values. The contents of these files are handled in the same way as other accounting profiles: After being populated over a configurable period of time or until the contents reach a configurable size, the files can be transferred automatically off the router to a predetermined server.

When using op scripts to enhance capacity planning, the scope of RMON alarm and accounting features can be expanded substantially and in doing so, users can locally monitor, threshold, and trend virtually any performance parameters normally available via the CLI.

Capacity Planning and Monitoring Examples

The following examples use pseudocode to illustrate basic scripts for capacity planning. Complete algorithms for each script are provided in the Appendix.

1. Setting and Monitoring an MIB Variable via RMON

This example shows how users can:

- Set up an event policy to trigger an operation script every 60 seconds.
- Use an op script to retrieve incoming traffic statistics from the specified interface and populate them into an MIB variable.
- Set up an RMON policy to monitor the MIB variable and send out an SNMP trap for threshold violations.

The following event policy triggers once every 60 seconds and executes the op script `write-jnxUtil.slax`:

```
.....
event-options {
  generate-event {
    IF_STATS_TRIGGER time-interval 60;
  }
  policy monitorTraffic {
    events IF_STATS_TRIGGER;
    then {
      event-script write-jnxUtil.slax {
        arguments {
          interface $int;
        }
      }
    }
  }
}
```

```
        user-name regress;
        output-filename eventlog.log;
        destination eventlog;
    }
}
```

.....
The op script `write-jnxUtil.slax` then extracts relevant interface information and populates the MIB variable:
.....

```
match / {
  <op-script-output> {
    var $rpc = {
      <get-interface-information> {
        <interface-name> $interface;
      }
    }
    var $out = jcs:invoke($rpc);
    call set_mib($out);
  }
}
template set_mib($out){
  // get relevant information
  var $rpc = <request-snmp-utility-mib-set> {
    <object-type> $object-type;
    <object-value> $object-value;
    <instance> $instance;
  }
  var $out1 = jcs:invoke($rpc);
  expr "value_" _ $object-value;
}
}
```

.....
The RMON policy below monitors the MIB variable and sends out SNMP traps for threshold violations:
.....

```
rmon {
  alarm 1 {
    description "rmon on jnxUtil";
    interval 10;
    falling-threshold-interval 30;
    variable .1.3.6.1.4.1.2636.3.47.1.1.3.1.2.105.110.112.117.116.95.98.112.115;
    sample-type absolute-value;
    rising-threshold 100;
    falling-threshold 5;
    rising-event-index 2;
    falling-event-index 2;
  }
}
```

.....
See the Appendix for complete examples and instructions for using this feature.
.....

2. Populating Custom Data into an Accounting File for Periodic Transfer and Processing

This example shows how to retrieve Packet Forwarding Engine (PFE) performance statistics periodically and write them to an accounting file.

First, create an event policy that periodically triggers an op script:

```
.....  
event-options {  
  generate-event {  
    ETHER_STATS_TRIGGER time-interval 60;  
  }  
  policy ether-stats {  
    events ETHER_STATS_TRIGGER;  
    then {  
      event-script op-ether-stats.slax {  
        user-name regress;  
        output-filename ether-stats.out;  
        destination ether-stats.out;  
        output-format xml;  
      }  
    }  
  }  
}
```

Next, configure the op script to populate the custom data into an accounting file:

```
.....  
var $rpc = <add-accounting-file-record> {  
  <file> $file;  
  <layout> $layout;  
  <fields> $fields;  
  <data> $data;  
}
```

See the Appendix for complete examples and instructions for using this feature.

Scripting Best Practices

Juniper Networks engineers recommend the following best practices for JUNOScript Automation:

- Document design decisions in each script. This makes the script the source for both design decisions and implementation logic. Incorporating these decisions will also be useful to future operators running the script.
- Arrange a peer review of the scripting code to be sure that the implementation is correct and the code is maintainable.
- Reuse the network's repository of scripting code. This reduces duplication and greatly simplifies the task of building complex scripts from simple ones.

Conclusion

JUNOScript Automation introduces an intelligent interface between human and network device to guard against configuration errors and to automate operational tasks. JUNOScript Automation reflects the customer's own business needs and procedures. The scripts are written by each network's experienced engineers—those who can flag potential errors in critical configuration elements such as routing and peering. The scripts also allow network engineers to set up early warning systems that not only detect emerging problems but also take immediate steps to avert further outages and restore normal operations.

References

The following online guides provide further information about JUNOS automation. All are linked from <http://www.juniper.net/techpubs/software/junos/>:

- JUNOS Configuration and Diagnostic Automation Guide
- JUNOS XML API Configuration Reference
- JUNOS XML API Guide
- JUNOS XML API Operational Reference

Appendix

The following sections include complete algorithms for the pseudocode examples provided earlier in this paper. Additional JUNOScript Automation documentation is cited in the References section above.

Commit Script Boilerplate

This example presents a basic SLAX script boilerplate for commit scripts. The boilerplate must be included as the starting point for all commit scripts that users create.

```
.....
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.slax";

match configuration {
/*
* Insert your code here.
*/
}
.....
```

Commit Script Examples

1. Imposing a Minimum MTU Setting

The example below determines the MTU of SONET/SDH interfaces, reports back if the MTU is less than the value of the `$min-mtu` variable, here set to 2048, and causes the commit operation to fail.

This example presents the complete MTU script; a pseudocode example is shown earlier in this paper.

```
.....
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.slax";

param $min-mtu = 2048;

match configuration {
  for-each (interfaces/interface[starts-with(name, 'so-') and mtu and
                                mtu < $min-mtu]) {
    <xnm:error> {
      call jcs:edit-path();
      call jcs:statement($dot = mtu);
      <message> {
        expr "SONET interfaces must have a minimum MTU of ";
        expr $min-mtu;
        expr ".";
      }
    }
  }
}
.....
```

Testing the MTU Commit Script

Follow these steps to test the preceding script:

- Copy the SLAX script into a text file and name the file `ex-so-mtu.slax`. Copy the `ex-so-mtu.slax` file to the router's `/var/db/scripts/commit` directory.
- Select the following configuration, and press Ctrl+c to copy it to the clipboard:

```
.....  
system {  
    scripts {  
        commit {  
            file ex-so-mtu.slax;  
        }  
    }  
}  
interfaces {  
    so-1/2/2 {  
        mtu 2048;  
    }  
    so-1/2/3 {  
        mtu 576;  
    }  
}
```

- ```
.....
```
- Merge the configuration into the routing platform configuration by issuing a `load merge terminal` configuration mode command:

```
.....
[edit]
user@host# load merge terminal
Type ^d at a new line to end your input.
```

Paste the contents of the clipboard as follows:

- At the command prompt, paste the contents of the clipboard.
- Press Enter.
- Press Ctrl+d.

- ```
.....
```
- Enter a commit command. The following output will appear:

```
.....  
[edit]  
user@host# commit  
[edit interfaces interface so-1/2/3]  
'mtu 576;'  
SONET interfaces must have a minimum MTU of 2048.  
error: 1 error reported by commit scripts  
error: commit script failure  
.....
```

2. Controlling LDP Configuration

This example script looks for interfaces configured at either the [edit protocols ospf] or the [edit protocols isis] hierarchy level, but not at the [edit protocols ldp] hierarchy level. If LDP is not enabled on the routing platform, there is no problem; otherwise, a warning is emitted with the message that the interface does not have LDP enabled.

The example below presents the complete code for the LDP commit script. A pseudocode example is shown earlier in this paper.

```

.....
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.slax";
match configuration {
  var $ldp = protocols/ldp;
  var $isis = protocols/isis;
  var $ospf = protocols/ospf;

  if ($ldp) {
    for-each ($isis/interface/name | $ospf/area/interface/name) {
      var $ifname = .;

      if (not(..apply-macro[name = 'no-ldp']) and
          not($ldp/interface[name =
              $ifname])) {
        <xnm:warning> {
          call jcs:edit-path();
          call jcs:statement();
          <message> "ldp not enabled for this interface";
        }
      }
    }
  }
  for-each (protocols/ldp/interface/name) {
    var $ifname = .;

    if (not(apply-macro[name = 'no-igp']) and
        not($isis/interface[name =
            $ifname]) and not($ospf/area/interface[name =
            $ifname])) {
      <xnm:warning> {
        call jcs:edit-path();
        call jcs:statement();
        <message> {
          expr "ldp-enabled interface does not have ";
          expr "an IGP configured";
        }
      }
    }
  }
}
.....

```

3. Creating a Complex Configuration Based on a Simple Interface Configuration

This example uses a macro to automatically expand a simple interface configuration. The script generates a transient change that assigns a default encapsulation type, configures multiple routing protocols on the interface, and applies multiple configuration groups.

The example shows the complete commit script code. A pseudocode example is shown earlier in this paper.

```

.....
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.slax";
match configuration {
    var $top = .;

    for-each (interfaces/interface/apply-macro[name = 'params']) {
        var $description = data[name = 'description']/value;
        var $inet-address = data[name = 'inet-address']/value;
        var $encapsulation = data[name = 'encapsulation']/value;
        var $isis-level-1 = data[name = 'isis-level-1']/value;
        var $isis-level-1-metric = data[name = 'isis-level-1-metric']/value;
        var $isis-level-2 = data[name = 'isis-level-2']/value;
        var $isis-level-2-metric = data[name = 'isis-level-2-metric']/value;
        var $ifname = ../name _ '.0';
        <transient-change> {
            <interfaces> {
                <interface> {
                    <name> ../name;
                    <apply-groups> {
                        <name> "interface-details";
                    }
                    if ($description) {
                        <description> $description;
                    }
                    <encapsulation> {
                        if (string-length($encapsulation) > 0) {
                            expr $encapsulation;
                        }
                        else {
                            expr "cisco-hdlc";
                        }
                    }
                    <unit> {
                        <name> "0";
                        if (string-length($inet-address) > 0) {
                            <family> {
                                <inet> {
                                    <address> $inet-address;
                                }
                            }
                        }
                    }
                }
            }
        }
        <protocols> {
            <rsvp> {
                <interface> {
                    <name> $ifname;

```



```

    }
  }
}
interfaces {
  so-1/2/3 {
    apply-macro params {
      description "Link to Hoverville";
      encapsulation ppp;
      inet-address 10.1.2.3/28;
      isis-level-1 enable;
      isis-level-1-metric 50;
      isis-level-2-metric 85;
    }
  }
}

```

.....

Merge the configuration into the router's configuration by issuing a `load merge terminal configuration mode` command:

.....

[edit]

.....

user@host# **load merge terminal**

Type ^d at a new line to end your input.

Paste the contents of the clipboard:

- a. At the command prompt, paste the contents of the clipboard.
 - b. Press Enter.
 - c. Press Ctrl+d.
-

- Enter a `commit` command.
-

[edit]

user@host# **commit**

.....

- When the user issues the `show interfaces | display commit-scripts | display inheritance` configuration mode command, the following output appears:
-

[edit]

user@host# **show interfaces | display commit-scripts | display inheritance**

```

so-1/2/3 {
  apply-macro params {
    clocking internal;
    description "Link to Hoverville";
    encapsulation ppp;
    inet-address 10.1.2.3/28;
    isis-level-1 enable;
    isis-level-1-metric 50;
    isis-level-2-metric 85;
  }
  description "Link to Hoverville";
}
##

```

```

## 'internal' was inherited from group 'interface-details'
##
clocking internal;
encapsulation ppp;
unit 0 {
    family inet {
        address 10.1.2.3/28;
    }
}
}

```

- When the user issues the `show protocols | display commit-scripts` configuration mode command, the following output appears:

```

[edit]
user@host# show protocols | display commit-scripts
rsvp {
    interface so-1/2/3.0;
}
isis {
    interface so-1/2/3.0 {
        level 1 {
            enable;
            metric 50;
        }
        level 2 metric 85;
    }
}
ldp {
    interface so-1/2/3.0;
}

```

4. Controlling a Dual Routing Engine Configuration

The examples below show how commit scripts can be used to simplify and control the configuration of dual RE platforms. The examples present complete code for each RE; pseudocode examples appear earlier in this paper.

Script 1: `ex-dual-re.slax`

```

version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.slax";
match configuration {
    for-each (system/host-name | interfaces/interface/unit/family/inet/address
            | interfaces/interface[name = 'fxp0']) {
        if (not(@junos:group) or not(starts-with(@junos:group, 're'))) {
            <xnm:warning> {
                call jcs:edit-path($dot = ..);
                call jcs:statement();
                <message> {
                    expr "statement should not be in target";
                    expr " configuration on dual RE system";
                }
            }
        }
    }
}

```

```
        }
    }
}
}
}

Script 2: ex-dual-re2.slax
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.slax";
match configuration {
    var $hn = system/host-name;

    if ($hn/@junos:group) {
    }
    else if ($hn) {
        <transient-change> {
            <groups> {
                <name> "re0";
                <system> {
                    <host-name> $hn _ '-RE0';
                }
            }
            <groups> {
                <name> "re1";
                <system> {
                    <host-name> $hn _ '-RE1';
                }
            }
            <system> {
                <host-name inactive="inactive">;
            }
        }
    }
    else {
        <xnm:error> {
            <message> "Missing [system host-name]";
        }
    }
}
}
```

Operation Script Examples

1. Restarting an FPC

This example restarts a Flexible PIC Concentrator (FPC) and modifies the output of the `request chassis fpc` command slightly to include the number of the restarting FPC.

The example presents the complete code for the FPC restart script. A pseudocode example is shown earlier in this paper.

```
.....  
version 1.0;  
  
ns junos = "http://xml.juniper.net/junos/*/junos";  
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";  
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";  
  
import "../import/junos.slax";  
  
var $arguments = {  
  <argument> {  
    <name> "slot";  
    <description> "Slot number of the FPC";  
  }  
}  
param $slot;  
  
match / {  
  <op-script-results> {  
    var $restart = {  
      <command> 'request chassis fpc slot ` _ $slot _ ` restart';  
    }  
    var $result = jcs:invoke($restart);  
    <output> {  
      expr "Restarting the FPC in slot ";  
      expr $slot;  
      expr ". ";  
      expr "To verify, issue the \"show chassis fpc\" command.";  
    }  
  }  
}
```

```
.....
```

Testing the FPC Restart Script

Follow these steps to test the preceding script:

- Copy the script into a text file and name the file `ex-fpc.slax`. Copy the `ex-fpc.slax` file to the router's `/var/db/scripts/op` directory.
- Include the statement file `ex-fpc.slax` at the `[edit system scripts op]` hierarchy level.

```
[edit system scripts op]
file ex-fpc.slax;
```

- Issue a `commit and-quit` command.
- When the `op ex-fpc slot number` command is entered, the following output appears:

```
user@host> op ex-fpc slot 0
Restarting the FPC in slot 0. To verify, issue the "show chassis fpc" command.
```

2. Customizing Show Interfaces Terse Output

The script below can be used to customize the output of the `show interfaces terse` command. The example shows the complete code for customizing the command; pseudocode examples are shown earlier in this paper.

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.slax";

var $arguments = {
  <argument> {
    <name> "interface";
    <description> "Name of interface to display";
  }
  <argument> {
    <name> "protocol";
    <description> "Protocol to display (inet, inet6)";
  }
}
param $interface;
param $protocol;

match / {
  <op-script-results> {
    var $rpc = {
      <get-interface-information> {
        <terse>;
        if ($interface) {
          <interface-name> $interface;
        }
      }
    }
    var $out = jcs:invoke($rpc);
    <interface-information junos:style="terse"> {
      if ($protocol='inet' or $protocol='inet6' or $protocol='mpls' or
          $protocol='tnp') {
        for-each ($out/physical-interface/
```

```

        logical-interface[address-family/address-family-name =
            $protocol] {
            call intf();
        }

    } else if ($protocol) {
        <xnm:error> {
            <message> {
                expr "invalid protocol: ";
                expr $protocol;
            }
        }
    }

    } else {
        for-each ($out/physical-interface/logical-interface) {
            call intf();
        }
    }
}
}
}
intf () {
    var $status = {
        if (admin-status='up' and oper-status='up') {

        } else if (admin-status='down') {
            expr "offline";
        } else if (oper-status='down' and ../admin-status='down') {
            expr "p-offline";
        } else if (oper-status='down' and ../oper-status='down') {
            expr "p-down";
        } else if (oper-status='down') {
            expr "down";
        } else {
            expr oper-status _ '/' _ admin-status;
        }
    }
    var $desc = {
        if (description) {
            expr description;
        } else if (../description) {
            expr ../description;
        }
    }
    <logical-interface> {
        <name> name;
        if (string-length($desc)) {
            <admin-status> $desc;
        }
        <admin-status> $status;
        if ($protocol) {
            copy-of address-family[address-family-name = $protocol];
        } else {
            copy-of address-family;
        }
    }
}

```

```

    }
  }
}

```

Capacity Planning and Monitoring Examples

The examples below present complete algorithms for the capacity monitoring scripts. Pseudocode examples are shown earlier in this paper.

1. Setting and Monitoring a MIB Variable via RMON

This example shows how users can:

- Set up an event policy to trigger an operation script every 60 seconds.
- Use the op script to retrieve incoming traffic statistics from the specified interface and populate them into an MIB variable.
- Set up an RMON policy to monitor the MIB variable and send out an SNMP trap for threshold violations.

The op script write-jnxUtil.slax extracts relevant interface information and populates the MIB variable:

```

version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns ext = "http://xmlsoft.org/XSLT/namespace";
import "../import/junos.xsl";

var $arguments = {
  <argument> {
    <name> "interface";
    <description> "Name of interface to display";
  }
}
param $interface;
match / {
  <op-script-output> {
    var $rpc = {
      <get-interface-information> {
        <interface-name> $interface;
      }
    }
    var $out = jcs:invoke($rpc);
    call set_mib($out);
  }
}
template set_mib($out){
  var $object-type = "integer"; /* can be a 'counter' as well */
  var $instance = "input_bps";
  var $object-value = $out/physical-interface/traffic-statistics/input-bps;
  var $rpc = <request-snmp-utility-mib-set> {
    <object-type> $object-type;
    <object-value> $object-value;
    <instance> $instance;
  }
  var $out1 = jcs:invoke($rpc);
  expr "value_" _ $object-value;
}

```

Testing the RMON Script

Follow these steps to test the preceding script:

- Copy the entire SLAX script above into a text file, and name the file `write-jnxUtil.slax`. Copy the file to the `/var/db/scripts/op` directory on the routing platform.
- Include the statement file `write-jnxUtil.slax` at the `[edit system scripts op]` hierarchy level.
- Configure the router as follows. Be sure to:
 - Replace `$int` in the configuration with the name of the interface being monitored, for example `ge-0/0/1`.
 - Change the user name "regress" under the "event-script" configuration to a valid userID on the system.

```

.....
system {
  scripts {
    op {
      file write-jnxUtil.slax;
    }
  }
}
event-options {
  generate-event {
    IF_STATS_TRIGGER time-interval 60;
  }
  policy monitorTraffic {
    events IF_STATS_TRIGGER;
    then {
      event-script write-jnxUtil.slax {
        arguments {
          interface $int;
        }
        user-name regress;
        output-filename eventlog.log;
        destination eventlog;
      }
    }
  }
  destinations {
    eventlog {
      archive-sites {
        /var/tmp;
      }
    }
  }
}
.....

```

- Configure RMON to monitor the MIB value. Be sure to:
 - Set up the NMS so that it receives the trap at the address configured in the SNMP trap configuration.
 - Note that in the RMON configuration, variable 1.3.6.1.4.1.2636.3.47.1.1.3.1.2.105.110.112.117.116.95.98.112.115 is the OID of jnxUtilIntegerValue.

```

.....
snmp {
  trap-group trap {
    version v2;
    destination-port 1624;
    targets {
      192.168.65.228;
    }
  }
  rmon {
    alarm 1 {
      description "rmon on jnxUtil";
      interval 10;
      falling-threshold-interval 30;
      variable .1.3.6.1.4.1.2636.3.47.1.1.3.1.2.105.110.112.117.116.95.98.1 2.115;
      sample-type absolute-value;
      rising-threshold 100;
      falling-threshold 5;
      rising-event-index 2;
      falling-event-index 2;
    }
    event 2 {
      description "event for op";
      type log-and-trap;
      community trap;
    }
  }
}
.....

```

- Issue a `commit and-quit` command.
- Start sending traffic to the interface being monitoring (ping should work).
- Wait for a few minutes for the counter to cross the threshold set by RMON.
- Check to see if the SNMP trap receiver received the RMON trap, or check from the CLI as follows:

```

.....
show snmp rmon
'rising threshold \{100\} crossed',
'jnxUtilIntegerValue.105.110.112.117.116.95.98.112.115',
.....

```

2. Populating Custom Data into an Accounting File for Periodic Transfer and Processing

This example shows how to retrieve PFE performance statistics periodically and write them to an accounting file.

The op script `op-ether-stats.slax`, shown below, populates custom data into an accounting file:

```

.....
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns ext = "http://xmlsoft.org/XSLT/namespace";

import "../import/junos.xsl";

var $arguments = {
  <argument> {
    <name> "interface";
    <description> "FE interface for which to record stats";
  }
}

param $interface;
param $file = "opevents";
param $layout = "ether-stats";

match / {
  <op-script-results> {
    if ($interface) {
      call do-stats($interface);
    } else {
      var $rpc = <get-interface-information> {
        <terse>;
      }
      var $if = jcs:invoke($rpc);
      for-each ($if/physical-interface/name[starts-with(., "fe-")]) {
        call do-stats($interface = .);
      }
    }
  }
}

template do-stats($interface) {
  var $stats-raw = {
    call get-stats($interface);
  }
  var $stats = ext:node-set($stats-raw);

  if ($stats/etherStatsPkts64Octets) {
    var $fields = "ifname,"
      _ "etherStatsPkts64Octets,"
      _ "etherStatsPkts65to127Octets,"
      _ "etherStatsPkts128to255Octets,"
      _ "etherStatsPkts256to511Octets,"
      _ "etherStatsPkts512to1023Octets,"
      _ "etherStatsPkts1024to1518Octets,"
      _ "etherStatsPkts64Octets_TX,"
      _ "etherStatsPkts65to127Octets_TX,"
      _ "etherStatsPkts128to255Octets_TX,"
      _ "etherStatsPkts256to511Octets_TX,"
      _ "etherStatsPkts512to1023Octets_TX,"
  }
}

```

```

    _ "etherStatsPkts1024to1518Octets_TX";

var $data = $interface _ ","
    _ $stats/etherStatsPkts64Octets _ ","
    _ $stats/etherStatsPkts65to127Octets _ ","
    _ $stats/etherStatsPkts128to255Octets _ ","
    _ $stats/etherStatsPkts256to511Octets _ ","
    _ $stats/etherStatsPkts512to1023Octets _ ","
    _ $stats/etherStatsPkts1024to1518Octets _ ","
    _ $stats/etherStatsPkts64Octets_TX _ ","
    _ $stats/etherStatsPkts65to127Octets_TX _ ","
    _ $stats/etherStatsPkts128to255Octets_TX _ ","
    _ $stats/etherStatsPkts256to511Octets_TX _ ","
    _ $stats/etherStatsPkts512to1023Octets_TX _ ","
    _ $stats/etherStatsPkts1024to1518Octets_TX;

<output> $fields;
<output> $data;

var $rpc = <add-accounting-file-record> {
    <file> $file;
    <layout> $layout;
    <fields> $fields;
    <data> $data;
}

var $res = jcs:invoke($rpc);
copy-of $res;

} else if ($stats/xnm:error) {
    copy-of $stats;

} else {
    <xnm:error> {
        <message> "statistics could not be found";
    }
}
}

template get-stats ($interface) {
    var $if = jcs:regex("fe-([0-9]+)/([0-9]+)/([0-9]+)", $interface);

    if ($if[1]) {
        var $fpc = "fpc" _ $if[2];
        var $pic = $if[3];
        var $port = $if[4];
        var $rpc = <request-pfe-execute> {
            <target> $fpc;
            /*<target> "cfeb";*/
            <command> "show fe-pic " _ $pic _ " stats " _ $port;
        }
        var $result = jcs:invoke($rpc);
        var $lines = jcs:break-lines($result);

        for-each ($lines) {
            var $pattern = "(etherStatsPkts[to0-9]+Octets[_TX]*) += +([0-9]+)";
            var $res = jcs:regex($pattern, .);
            if ($res[1]) {
                <xsl:element name=$res[2]> $res[3];
            }
        }
    }
}

```

```

    } else {
      <xnm:error> {
        <message> "invalid interface name: "_ $interface_" or pfe name: "_ $fpc;
      }
    }
  }
}

```

Testing the PFE Script

Follow these steps to test the preceding script:

- Copy the SLAX script into a text file, and name the file `op-ether-stats.slax`. Copy the file to the `/var/db/scripts/op` directory on the routing platform.
- Include the statement `file op-ether-stats.slax` at the `[edit system scripts op]` hierarchy level.
- Configure an event policy to trigger the script every 60 seconds.
- Configure accounting options:

```

system {
  scripts {
    op {
      file op-ether-stats.slax;
    }
  }
}
event-options {
  generate-event {
    ETHER_STATS_TRIGGER time-interval 60;
  }
  policy ether-stats {
    events ETHER_STATS_TRIGGER;
    then {
      event-script op-ether-stats.slax {
        user-name regress;
      }
    }
  }
}
accounting-options {
  file opevents {
    size 256k;
    files 5;
    transfer-interval 10;
  }
}

```

- Send packets to the router and wait a few minutes until the accounting file is populated with PFE statistics.
- Check the accounting file on a Web server. The data will look like this:

```

% tail opevent
ether-stats,1178289363,fe-0/1/0,0,0,0,0,0,0,0,0,0,0,0,0,0
ether-stats,1178289364,fe-0/1/1,0,0,0,0,0,0,0,0,0,0,0,0,0
ether-stats,1178289364,fe-0/1/2,0,0,0,0,0,0,0,0,0,0,0,0,0
ether-stats,1178289365,fe-0/1/3,0,0,0,0,0,0,0,0,0,0,0,0,0

```

About Juniper Networks

Juniper Networks, Inc. is the leader in high-performance networking. Juniper offers a high-performance network infrastructure that creates a responsive and trusted environment for accelerating the deployment of services and applications over a single network. This fuels high-performance businesses. Additional information can be found at www.juniper.net.

Corporate and Sales Headquarters

Juniper Networks, Inc.
1194 North Mathilda Avenue
Sunnyvale, CA 94089 USA
Phone: 888.JUNIPER
(888.586.4737)
or 408.745.2000
Fax: 408.745.2100

APAC Headquarters

Juniper Networks (Hong Kong)
26/F, Cityplaza One
1111 King's Road
Taikoo Shing, Hong Kong
Phone: 852.2332.3636
Fax: 852.2574.7803

EMEA Headquarters

Juniper Networks Ireland
Airside Business Park
Swords, County Dublin,
Ireland
Phone: 35.31.8903.600
Fax: 35.31.8903.601

Copyright 2009 Juniper Networks, Inc. All rights reserved. Juniper Networks, the Juniper Networks logo, JUNOS, NetScreen, and ScreenOS are registered trademarks of Juniper Networks, Inc. in the United States and other countries. JUNOS is a trademark of Juniper Networks, Inc. All other trademarks, service marks, registered marks, or registered service marks are the property of their respective owners. Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

To purchase Juniper Networks solutions, please contact your Juniper Networks representative at 1-866-298-6428 or authorized reseller.

