Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Bachelor of Science in Engineering -
Computer Networks 15.0 credits

AUTOMATED DEPLOYMENT OF
CUSTOMER-PREMISES EQUIPMENT

Christian Åberg
cag12003@student.mdh.se

Examiner: Mats Björkman
Mälardalen University, Västerås, Sweden

Supervisor: Nils Müllner
Mälardalen University, Västerås, Sweden

Company supervisor: Jörgen Jonasson,
TDC, Stockholm

June 20, 2016
Abstract

Zero touch deployment tools perform installation and configuration of computer networking devices without human interaction. Modern zero touch deployment tools generally lack support for routers and are platform dependent. This forces Internet Service Providers to rely on manual node deployment methods when deploying Customer-Premise Equipment which is time consuming and error prone. This thesis investigates how the process of deploying Customer-Premises Equipment can be automated and optimized. State-of-the-art is discussed to point out requirements and possible improvements. Based on that, a tool is implemented performing deployment of Customer-Premises Equipment while minimizing manual intervention. As Internet Service Providers typically work with equipment from various vendors, multivendor support is implemented. This thesis studies common methods for initial Customer-Premises Equipment configuration. Request for Comments and Technical manuals are consulted regarding network protocols, zero touch deployment tools and router operating systems.
# Table of Contents

1 **Introduction**  
1.1 Research Objectives ............................................. 1  
1.2 Motivation .................................................. 1  
1.3 Method .................................................. 2  

2 **Background**  
2.1 Node Deployment Concepts ........................................... 3  
2.1.1 Request for Comments ......................................... 3  
2.1.2 Computer Networking Basics ................................. 3  
2.1.3 Operating System Management ........................... 5  
2.1.4 Configuration Management ............................... 6  
2.1.5 Manual Node Deployment Methods ......................... 7  
2.2 Related Work ............................................... 8  
2.2.1 Available Node Configuration Tools ......................... 8  
2.2.2 Zero Touch Deployment Methods .......................... 9  
2.2.3 Cisco Smart Install ..................................... 9  
2.2.4 Juniper Zero Touch Provisioning ......................... 11  
2.3 Manual Node Deployment Study .............................. 12  

3 **Platform Independent Deployment System (PIDS)**  
3.1 Architecture ............................................... 14  
3.2 Kernel .................................................. 15  
3.3 API .................................................. 16  
3.4 User Interface ............................................ 18  
3.5 Platform Drivers ........................................... 21  
3.5.1 Profiles ............................................. 21  
3.5.2 Kernel Instructions ..................................... 23  
3.5.3 Console Cable Communication ............................ 24  
3.6 Console Cable Execution Methods ............................ 24  

4 **Analyzing PIDS** ............................................. 27  
4.1 Design .................................................. 27  
4.2 Test 1: Single Deployment .................................... 29  
4.3 Test 2: Dual Deployment .................................... 30  
4.4 Test 3: Triple Deployment .................................. 31  

5 **Interpretation** ............................................... 32  
5.1 PIDS Performance .......................................... 32  
5.2 PIDS versus Zero Touch Deployment Methods ............ 33  

6 **Conclusion** .................................................. 35  

References .................................................. 36  

Appendix A Initial Configuration Study Manuals ................. 37  
Appendix B PIDS Source Code Reference ........................ 39
List of Figures

1. Topology icons ................................................. 4
2. Network topology with device connections ................... 5
3. Configuration sent through a Console cable ................. 6
4. Configuration sent over SSH ................................ 7
5. Configuration sent over an IP network ....................... 7
6. A Console cable for network node configuration .......... 7
7. Example topology for Cisco Smart Install .................. 10
8. Example topology for Juniper Zero Touch Provisioning .... 12
9. Device roles in the PIDS architecture ...................... 14
10. Software architecture within the PIDS Server ............. 15
11. Concurrent deployment of two Cisco routers ............... 16
12. Concurrent deployment of two Juniper routers .......... 16
13. Client connecting to a server through PIDS API .......... 18
14. The Home-view in PIDS GUI .............................. 18
15. Adding processes in PIDS GUI ............................ 19
16. Viewing active processes in PIDS GUI .................... 19
17. Analyzing devices in PIDS GUI .......................... 20
18. Selecting setup profile for devices in PIDS GUI .......... 20
19. Hold Timer operation in HBEM communication ............. 25
20. Demonstration of Expect Based Execution Method ......... 26
21. The topology when analyzing PIDS ......................... 27
22. PIDS analysis - Topology for test 1 ....................... 29
23. PIDS analysis - Topology for test 2 ....................... 30
24. PIDS analysis - Topology for test 3 ....................... 31
25. Average deployment time (no system image upgrades) .... 32
26. Average deployment time (with system image upgrades) .. 32
27. PIDS two step deployment procedure ....................... 33
28. S1 experiences failure and must be replaced ............. 34
29. S1_new replaces S1 and is automatically configured by Cisco Smart Install ... 34

List of Tables

1. Configuration methods for network nodes .................. 8
2. Functions separated from the Startup Configuration in Cisco switches .................. 11
3. Available Manual Node Deployment Methods per platform .. 13
4. Required variables for processes within the PIDS kernel . 15
5. PIDS kernel process list .................................. 16
6. PIDS kernel process list after modification ............... 16
7. Request-packet types in PIDS API ......................... 17
8. Drivers methods that must be callable by the kernel .... 23
9. Methods available for the yost object .................... 24
10. Test result of analysis 1A .................................. 29
11. Test result of analysis 1B .................................. 29
12. Test result of analysis 2A .................................. 30
13. Test result of analysis 2B .................................. 30
14. Test result of analysis 3A .................................. 31
15. Test result of analysis 3B .................................. 31
16. Comparison of PIDS to Zero Touch Deployment Methods .. 33
17. Manuals for the devices in the Initial Configuration Study .... 37
1 Introduction

A demarcation point is commonly understood by Internet Service Providers (ISP) as a point at which a ISP network end and a customer network begin. If an error occur within the customer side of the demarcation point, the customers is responsible to fix the error and vice versa.

ISP customers can generally choose the demarcation point placement. Due to its administrative simplicity, many customers choose to exclude the Internet connected routers (Customer-Premises Equipment) from the customer side of the demarcation point, thus handing over the equipment responsibility to the ISP. This responsibility includes:

- configuration,
- deployment,
- monitoring and
- troubleshooting of the Customer-Premises Equipment.

Zero Touch Deployment Methods automate deployment of networking equipment. Cisco and Juniper are two large actors in the networking equipment domain that have developed their own implementation of Zero Touch Deployment Methods for switches.

Switches (in their simplest form) provide end users with connectivity to a single computer network, as opposed to routers that interconnect multiple computer networks. Routers also have support for various transmission media which is why routers often are selected as Customer-Premises Equipment.

1.1 Research Objectives

The process of deploying routers is not very streamlined with the absense of Zero Touch Deployment Methods. Deploying routers often involve manual configuration in various ways (Manual Node Deployment Methods). For this reason, this thesis studies the following objectives:

**Objective 1:** Determine how the process of deploying Customer-Premises Equipment can be improved. This is achieved by investigating the requirements for an automated deployment process and by reviewing state-of-the-art tools.

**Objective 2:** Develop a tool (based on the knowledge from **Objective 1**) to perform an automated deployment of Customer-Premises Equipment. The tool is referred to as the Platform Independent Deployment System (PIDS).

1.2 Motivation

Consider a scenario where an Internet Service Provider (ISP) has to configure and deploy a large number of Customer-Premises Equipment for their clients. Manually configuring the nodes is a time consuming and error prone process and the Zero Touch Deployment Methods available only work with switches and not with routers.

The world is becoming more and more automated; but how come there are no efficient tools for automating Customer-Premises Equipment deployment? No Zero Touch Deployment Methods with support for routers was found, despite thorough research. What node deployment tools exist today and how do they work?

A tool automating deployment of Customer-Premises Equipment is developed and presented in this thesis. Different routers have varied deployment methods and functionalities which leads to the following questions:

- How do routers handle system images and configurations?
- What are the methods of accessing unconfigured routers?
- What Zero-Touch Deployment Methods exist today?
- When not using a Zero-Touch Deployment Method, what Manual Deployment Methods exist?
1.3 Method

In order for PIDS to achieve platform independency, it is important to have knowledge in current deployment methods for networking equipment. This thesis studies the following two fields:

- Current Zero Touch Deployment Methods and

Current Zero Touch Deployment Methods are studied by reviewing documentation for common network operating systems (e.g. Cisco IOS or Junos OS). Current Manual Node Deployment Methods are studied by reviewing the documentation for various networking equipment. A study is performed to determine common methods when manually deploying unconfigured Customer-Premises Equipment from various vendors.

PIDS development starts after studying current Zero Touch Deployment Methods and Manual Node Deployment Methods. Finally, a study is conducted to determine the performance of PIDS.
2 Background

At times, network technicians and engineers encounter situations when a large amount of network nodes need to be configured with a standard template either before deployment or during upgrades. Typical scenarios include:

- **Access layer node deployment**: The access layer of a campus network architecture connects people (End Users) to an IP network. The node quantity requirements expand with the amount of end users.

- **Pre-configuration of Customer-Premises Equipment**: Internet Service Providers provide customers with pre-configured equipment. Manual configuration of Customer-Premises Equipment is prone to errors.

- **Network node upgrades**: Network nodes need upgrading. Possible reasons are operating system patches or configuration changes. The need for network node upgrades embodies each layer of the network design architecture and can include a large number of devices.

Each scenario described above can be solved through manual configuration. Manual configuration of production equipment is both time consuming and prone to human errors.

According to Jörgen Jonasson, Senior Network Architect at TDC1, some of these issues are addressed by creating configuration templates before deployment. Before the configuration can be transferred to a device, the system image of a network node needs to be adopted to accommodate the configuration template. The configuration template process is described below:

1. Identify the platform.
2. Check if the system image is compatible with the configuration template. If so, go to Step 3.
   (a) Install a compatible system image to the device.
   (b) Restart the network node (if necessary).
3. Transfer the configuration template to the device.
4. Manually execute commands that could not be included in the configuration template.
5. Save the active configuration on the network node.

This process is repeated for each network node. Several current Zero Touch Deployment Methods automate deployment of network switches; but no such tool was found for routers despite through research.

2.1 Node Deployment Concepts

The following sections investigates some basic concepts composing the area of node deployment.

2.1.1 Request for Comments

Request for Comments (RFC) are a type of technical report describing functions and protocols for the Internet and computer networks (e.g. DHCP and RADIUS)2.

2.1.2 Computer Networking Basics

Computer networks allow computers and embedded systems to transmit data through a common infrastructure. They consist of nodes contributing specific functions to the network. These functions include:

- Delivering data.

---

1May 2016 - [http://tdc.se/](http://tdc.se/)

2March 2016 - [https://www.ietf.org/rfc.html](https://www.ietf.org/rfc.html)
• Analyzing data for security threats.
• Preventing data from accessing specific resources.
• Limiting data-throughput for individual nodes.

The computer networking field has a common terminology to describe the roles. Most of the terminology is standardized with RFCs. The following paragraphs describes the terminology relevant to this thesis:

IP Addresses: Devices participating in computer networks are assigned a unique IP-address. IP-addresses allow devices to initiate private data-flows between one another. If two devices are assigned the same IP-address, this is called an IP conflict and communication might be interrupted. There are several versions of IP and each version differs in functionality and amount of addresses.[1]

IP Subnets: An IP Subnet is a range of IP addresses. If an ISP has ownership of an IP subnet, the range can be further divided into smaller subnets that the ISP can share with its customers. The Internet is divided into millions of subnets [2].

Router: Routers forward data between subnets. If a computer in one subnet needs to reach a device whose IP address is located in another subnet, the computer will forward the packet to a router. The router will make a decision on how to forward the data (this process is called routing). Routing will generally lead to one of three things:

- The router forwards the packet directly to the destination device.
- The router forwards the packet to another router that is between itself and the receiver.
- The router has nowhere to forward the data, discards it and optionally returns a ICMP Destination Unreachable to the sender.

Routing-decisions are generally based on sophisticated algorithms (Routing Protocols). Routers are the main building block of the Internet [3].

Switch: A switch performs data forwarding within a subnet. Switches forwards data based on MAC addresses instead of IP addresses.

Firewall: A firewall performs data analysis and is able to prevent intrusions to a private network. The firewall possesses a database of known vulnerabilities which is compared to analyzed data. If the data has a match in the database, the firewall discards the packet and optionally notifies an administrator [4].

![Figure 1: Topology icons.](image-url)
Network Topologies The physical connections in a Computer Network can be graphically represented in a Network Topology. Figure 1 shows some icons for common devices in computer networks. Figure 2 shows how topologies can be help describing networks with (in this case) eight nodes and eleven interconnections:

The cable types connecting the nodes can also be determined by reviewing the topology. In Figure 2, it is shown that the connection between the Customer-Premises Equipment and the ISP are of a different type than the rest of the topology.

There are no current standardizations addressing what icons to use in topologies. Instead, the networking company Cisco has published a set of icons which has been widely accepted today.

2.1.3 Operating System Management

The operating system (OS) in a router provides for:

- Resource Management
- Hardware Drivers
- Node Functionality

Without an operating system, a router is not able to function. High-end Customer-Premises Equipment are embedded systems with specialized hardware to perform routing. Thus, the CPU is not intensively interrupted during high data loads. The hardware performs routing-decisions based on previous instructions received by the OS [5].

The OS is crucial to a routers performance and it is therefore important to run an OS free from bugs in production environments. It is generally a good practice to update the operating system when new versions of the software become available. Vendors have different ways of managing the OS of their devices. It is therefore important to review the platform documentation before managing the operating system of a device. After upgrading the OS of a router, a couple of unexpected events can occur:

- Functions have been removed
- Syntax has been changed
- New bugs occur

The old configuration might therefore be incompatible to the new OS. It is important for the administrator to consider these events before upgrading production nodes. A downgrade with respect to the points above can also be considered as a good alternative of maintaining a stable production network.

2.1.4 Configuration Management

This section presents alternative ways to manage configurations in Customer-Premises Equipment.

Configuration Locations  How a network node handles configurations depends on the active operating system. There are multiple approaches vendors can take where simplicity is weight against functionality. The list below presents various ways nodes handle configurations:

- **Single file**: This is the simplest method from an administrators point of view. Only the single configuration file is read by the operating system and loaded into memory during node boot up.

- **Multiple files**: Some devices divide the configuration into multiple files. In these cases, there is often a main configuration file storing the most general parts of the devices configuration. The administrator needs to be aware of the file structure when performing configuration management, which adds a layer of complexity.

- **Database**: Databases provide additional functionality to the configuration management. Databases generally consist of a single binary file. Thus, the configuration structure is defined within the database file instead of the devices file system. The configuration database records changes over time and provides the administrator with an option to return to a previous version of the configuration. Cisco refers to this process as a *Configuration Rollback* [6, p.4-81].

It is necessary to determine how a device handles configurations before performing Configuration Management.

Applying Configurations  There are generally two ways of transferring a configuration to a node:

- Configuration through a Console Cable.
- Configuration through a remote management protocol.
- Transferring a configuration file from a remote file server.

All three methods are described below:

**Console Cable**  Configurations can be applied to a device through a Console Cable. Console Cable management is considered as Out-of-band because it is not carried out via an IP network. Figure 3 demonstrates a computer sending an configuration to a router through a console cable.

![Console Cable](image)

Figure 3: Configuration sent through a Console cable.

**Remote Management**  A device can be configured remotely over an existing IP network. Remote management protocols are considered In-band. Common protocols include:

- SSH
- Telnet
- SNMP
The device must have IP connectivity for remote management protocols to function. If for some reason the device loses connectivity as of misconfiguration, the device might never be reachable with remote management techniques again, until the misconfiguration is fixed by other means.

Figure 4 demonstrates how a person in Stockholm can configure a device in Oldenburg through remote management protocols.

![Figure 4: Configuration sent over SSH.](image)

**Remote File Server** Configurations can be transferred to a device from an IP based file server. This method is considered an in-band-management technique since the configuration is carried over an existing IP network. Common protocols for sending configurations include:

- HTTP
- FTP
- TFTP

Before a configuration can be sent remotely to a node, all participating devices require IP addresses and permissions to establish sessions between one another. Figure 5 demonstrates a scenario where a configuration is transferred to a device over an existing IP network:

![Figure 5: Configuration sent over an IP network.](image)

### 2.1.5 Manual Node Deployment Methods

Before network equipment can function in production environments, the equipment must be configured. For example:
• An administrator might need to activate remote management services for a device.
• An administrator needs to secure the node by setting an access password.

If the equipment has factory default settings, the techniques for accessing and configuring the devices vary in regard to the platform. Some devices require the administrator to access the device through a console cable (Figure 6) while others initially must be configured with remote management techniques such as HTTP and Telnet. Table 1 presents two common deployment methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI-based</td>
<td>Some devices require initial configuration to be performed remotely through a web browser or some specialized application. A pre-configured IP address is required on the device for it to be accessible for the administrator.</td>
</tr>
<tr>
<td>CLI-based</td>
<td>Some devices require initial configuration to be performed Out-of-band through a console cable.</td>
</tr>
</tbody>
</table>

Table 1: Configuration methods for network nodes.

Section 2.3 investigates what Manual Node Deployment Methods are most common for enterprise equipment as of March 2016.

2.2 Related Work
Several tools exist that automate different aspects of network management. Many vendors have developed their own proprietary configuration management tools for their products.

Cisco Smart Install and Juniper Zero Touch Provisioning respond well to the Problem Formulation and were therefore studied in more depth in Section 2.2.2.

2.2.1 Available Node Configuration Tools
The following section presents some of the available tools for automated node configuration. A brief discussion is included for each tool. The tools are listed in alphabetical order:

**Ansible**  Ansible performs automated configuration deployment. By grouping network nodes, Ansible can apply configuration changes to multiple devices at once. Ansible is open source and licensed under GPLv3.

Ansible connects to the nodes through remote management techniques. SSH is the default method. Thus, SSH needs to be manually activated on the nodes before Ansible can operate.

**Cisco Smart Install**  Cisco Smart Install is a vendor specific and proprietary tool performing automated image and configuration deployment on Cisco switches. The tool can not operate on non Cisco based platforms since it is based on the proprietary protocol Cisco Discovery Protocol (CDP). Thus, Cisco Smart Install is not suitable in multivendor environments.[7]

**Juniper Zero Touch Provisioning**  Juniper Zero Touch Provisioning performs automatic deployment of juniper switches without manual intervention. The client switches get informed (through DHCP options) about the location of an operating system image and a configuration file [8, p.63].

Juniper Zero Touch Provisioning is only applicable with Juniper switches. Considering that the methods behind Juniper Zero Touch Provisioning are based on DHCP options, other vendors could freely implement this solution [9].

---

5March 2016 - [http://www.ansible.com/configuration-management](http://www.ansible.com/configuration-management)
NetMRI  NetMRI is a node management system which specializes in network discovery, network change automation and configuration compliance management. NetMRI has support for both traditional- and virtual nodes (like Cisco VRF). NetMRI is a proprietary software licensed under the Infoblox End-User License Agreement\(^6\). The primary goal behind NetMRI is to provide a tool for performing change deployment and configuration compliance for preconfigured nodes. NetMRI is not intended for node deployment\(^7\).

SolarWinds Network Configuration Manager  SolarWinds Network Configuration Manager performs periodic configuration backups, configuration change monitoring and configuration change automation\(^8\). SolarWinds Network Configuration Manager is a proprietary software. The tool is intended for pre-configured environments and does not perform automatic node deployment.

Summary:  A lot of existing node configuration tools are developed to perform configuration management in pre-configured systems and networks. They provide for additional control in complex networks where node management is challenging. Most of these tools have no precise method to solve the problems as described in this thesis.

The most interesting of the tools in the scope of this thesis are Cisco Smart Install and Juniper Zero Touch Provisioning. Section 2.2.2 investigates how Cisco Smart Install and Juniper Zero Touch Provisioning perform node deployment.

2.2.2 Zero Touch Deployment Methods

Cisco Smart Install and Juniper Zero Touch Provisioning perform system image upgrades and configuration transfers. Both tools respond to the thesis problem formulation. Some shortages of the tools are:

- platform dependency,
- both tools are proprietary and
- they only work with switches.

The following sections investigate the technical concepts, methods and communication protocols of the tools.

2.2.3 Cisco Smart Install

Cisco Smart Install performs system image upgrades and configuration of nodes. Cisco Smart Install requires no manual intervention after a client is plugged into a Cisco Smart Install compatible network.

The device managing a Cisco Smart Install network is called a Director. The Director communicates directly to client switches. The client provides the Director with information regarding its:

- Main MAC address
- Product Identification

The Director provides the client with knowledge about where to find the System Image and the Configuration File. The client then initiates the upgrade. The Cisco Smart Install architecture defines several roles:

- **Client** - The Client identifies itself to the Director in order to determine if any predefined System Image and Configuration File exist.
- **Director** - The Director communicates with clients and provides them with TFTP paths for the System Image and the Configuration File.

\(^6\)March 2016 - https://www.infoblox.com/company/legal/eula-netmri
\(^7\)March 2016 - https://www.infoblox.com/products/network-automation/netmri
• **DHCP Server** - The DHCP Server provides the client with:
  – an IP address and
  – a destination IP to the TFTP Server.

• **TFTP Server** - The TFTP Server stores all system images and configuration files for the clients. A TFTP Server can be configured on the Director or on a remote server.

There are in total three communication protocols participating in the Cisco Smart Install architecture:

• **CDP** - *Cisco Discovery Protocol* is a Cisco proprietary communication protocol. CDP communicates through multicast addresses. Thus, no IP addresses are required on the nodes for the communication to function [10, p.321]. In Cisco Smart Install, the Director identifies the client with CDP.

• **DHCP** - *Dynamic Host Configuration Protocol* is a communication protocol that dynamically assigns IP addresses to network nodes. A DHCP Server can include additional variable data in its packages through DHCP Options [11]. In Cisco Smart Install, DHCP provide clients with IP addresses and a TFTP Server address. The Director inserts additional options into the Offer from the DHCP Server through DHCP Snooping [12, p.1-2]. The following options are inserted into the DHCP Offer by the Director [12, p.1-5]:
  – Option 67 - The Configuration File.
  – Option 125 - Vendor-Specific
    * Suboption 5 - The image list file
    * Suboption 16 - Director IP address

• **TFTP** - *Trivial File Transfer Protocol* is a communication protocol allowing clients to download and upload files to a central entity. TFTP uses connectionless transmission techniques (UDP) to request and deliver files [13].

In Cisco Smart Install, a TFTP Server stores the system images and configuration files for the clients. If a remote TFTP Server is configured, the Director will automatically upload a tailored configuration file and an image list file on the TFTP Server.

**Cisco Smart Install Example**  The following section describes how nodes communicate in a working Cisco Smart Install environment. The example is based on the topology as presented in Figure 7:

![Figure 7: Example topology for Cisco Smart Install.](image-url)
Configuring Cisco Smart Install Equipment  The Cisco Smart Install service must be configured on the Director. During configuration, the administrator selects a source IP address and interface for the Cisco Smart Install service to operate on.

The administrator must manually match device identifiers to a System Image and a Configuration File for each client in the network. If the network only consists of equipment of the same model, a default identifier can be configured. Otherwise, multiple device groups are configured to accommodate and serve various platforms.

The Director can optionally be configured as a DHCP Server. In the example above, a remote DHCP Server is configured. Option 150 must be included in DHCP Offers specifying the IP address of the TFTP Server.

Connecting Clients to the Network  Once all Cisco Smart Install equipment are set up, clients can be connected to the network. Given Client $x$ is a Cisco Smart Install capable switch with no present Startup Configuration in NVRAM. During startup, Client $x$ registers its platform and MAC address to the Director. The Director stores the client information in a local database. If the client identification matches a default policy or built-in group, the Director informs the client what System Image and Configuration File to download.

The client downloads the System Image and Configuration File from the TFTP Server. It is important for the client to have sufficient memory in order to hold the new System Image. Once the System Image and Configuration File are downloaded from the TFTP Server, the client will perform an automatic reboot to finalize the update.

Post-Install Operations  Not all parts of a Cisco Switch configuration are stored in the Startup Configuration. Thus, some manual additions of commands might be required to activate certain functionalities of the node [12, 2-38]. Table 2 describes functions that are not stored in the Startup Configuration in Cisco switches:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLAN</td>
<td>VLAN configurations and information are stored in the file <code>vlan.dat</code> [14, Section &quot;Viewing VLAN Information&quot;]</td>
</tr>
<tr>
<td>VTP</td>
<td>VTP configurations and information are also stored in the file <code>vlan.dat</code> [14, Section &quot;Filename and Location&quot;]</td>
</tr>
</tbody>
</table>

Table 2: Functions separated from the Startup Configuration in Cisco switches.

Some configurations that are stored in the Startup Configuration must be unique for each node. These configurations include:

- Hostname
- RSA Key-Pair

These configurations should not be applied in a standard Startup Configuration for a group of switches. Thus, they must be configured manually for each node.

Since release IOS 15.2(2)E it is possible to perform Post-install operations on individual network nodes. The Post-Install Operations are collected in a Post-Install Script (populated with Cisco IOS commands) that the node runs after standard installation.

2.2.4 Juniper Zero Touch Provisioning

Juniper Zero Touch Provisioning is a method to deploy Juniper switches with minimal manual intervention. Juniper Zero Touch Provisioning requires less services to operate than Cisco Smart Install and is only based on open protocols. The following protocols are used in a Juniper Zero Touch Provisioning environment: [8, p.63]

- DHCP
- FTP (optional)
The following description refers to Figure 8. The topology consists of three client devices (in need for an upgrade), a DHCP Server (which will provide the client devices with IP addresses and configuration paths) and a file server (which will store the System Image and the Configuration Files).

The client devices perform a Juniper Zero Touch Provisioning upgrade only if they have the default factory configuration installed. The administrator can manually install the factory configuration by executing the command `request system zeroize`.

Once the client has booted, the client broadcasts a DHCP Request to receive an IP address and paths to the System Image and Configuration File. The DHCP Server must be configured to include the following DHCP Options [8, p.68]:

- Option 43 - Vendor Specific Information
  - Suboption 00 - The name of the System Image file on the FTP, HTTP or FTP server.
  - Suboption 01 - The name of the Configuration File on the FTP, HTTP or FTP server.
  - Suboption 04 - Specifies the transfer mode used:
    * FTP
    * HTTP
    * TFTP

The client retrieves the necessary files for the upgrade, based on the information in the DHCP offer packet. The client compares the remote System Image to the System Image stored locally. If the images are of the same version and type, no download will be performed [8, p.66].

Once the System Image and Configuration File are downloaded from the file server, the switch will reboot to finalize the installation.

### 2.3 Manual Node Deployment Study

This thesis performs a study to determine what Manual Node Deployment Methods are most common for various Customer-Premises Equipment as of March 2016. The conclusion of the study is that most modern Customer-Premises Equipment must be initially configured through a Console cable before deployment. Remote Management techniques are not generally activated when the devices have only factory default setting.
Method

Only Customer-Premises Equipment are analyzed in the study. Vendors are selected to participate based on their current market share holdings in the networking field\(^9\). Equipment is selected based on the following criteria:

- The producer identifies the equipment as Customer-Premises Equipment.
- The devices are aimed at medium- to large-size enterprises.

Device manuals are reviewed to identify what initial deployment methods are available when no prior configuration has been applied the equipment. Appendix A provides a list of all the manuals selected for this study.

Result

The final result of the study is presented in Table 3. All of the analyzed devices support Console cable configuration out of the box. None of them can be configured with Remote Management techniques without any prior configuration to the device.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Series</th>
<th>Console cable</th>
<th>Browser/Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cisco</td>
<td>4000 Series</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Cisco</td>
<td>800 Series</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>HP</td>
<td>MSR3000 Router Series</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>HP</td>
<td>MSR4000 Router Series</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Juniper</td>
<td>LN Series</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Huawei</td>
<td>AR3200 Series Enterprise Routers</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Huawei</td>
<td>AR2200 Series Enterprise Routers</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Available Manual Node Deployment Methods per platform.

Conclusion

Initial configuration of enterprise Customer-Premises Equipment is generally performed through a console cable. None of the analyzed devices have support for Remote Management for initial configuration.

Discussion

None of the devices in the study can be initially configured with remote management techniques. This could potentially mean there is a general consensus among producers to avoid this method. Security is a main concern when it comes to remote management and this could potentially be a reason not to enable it per default.

Networking equipment aimed at small- and home offices on the other hand typically focuses more on simplicity and user-friendliness. This equipment is often accessed by a web browser or a specialized application. Small- and home office equipment is outside the scope of this thesis and is therefore not selected for this study.

3 Platform Independent Deployment System (PIDS)

The Platform Independent Deployment System (PIDS) is capable of configuring and installing multiple network nodes simultaneously. Although this thesis focuses on deployment of Customer-Premises Equipment, PIDS is designed to configure a variation of network devices. This section explains how PIDS performs node deployments. Architectural design choices are also discussed.

PIDS automates many deployment steps. The tool can not be considered a Zero-Touch Deployment Method since it must be managed by an administrator during deployment. The tool provides the following functionalities:

- PIDS performs the actual installation decreasing the required expertise from administrators,
- it supports simultaneous deployments, and
- perform installations based on pre-configured deployment strategies (profiles).

The source code is presented in Appendix B (PIDS Source Code Reference).

3.1 Architecture

Several device roles are defined within the PIDS architecture. Figure 9 demonstrates each role and how they interconnect to one-another.

- **PIDS Client:** The PIDS Client role provides end users with a graphical interface to the service. This role communicates to the PIDS Server through a socket-based API (PIDS API). Thus, the Client can run locally on the Server or remotely on a different host.

- **PIDS Server:** A PIDS Server communicates with Target Nodes through console cables. This role performs the actual installation and configuration of the nodes. The functionalities of the PIDS Server can be further divided into:
  - **Kernel:** The kernel interconnects various functionalities of the service.
  - **Drivers:** Drivers extend PIDS platform support. They communicate to Target Nodes and translates PIDS commands to the devices.

- **Target Node:** Networking equipment for which deployment will be performed are identified as Target Nodes. PIDS supports Target Nodes for various platforms.

The PIDS kernel allows for simultaneous deployments of multiple Target Nodes. For this to work, the PIDS server must have a console cable connection to multiple Target Nodes.
3.2 Kernel

PIDS is designed with modularity in mind to allow for future optimizations and improvements. System functionalities are separated from each other and interconnected through the kernel. Figure 10 demonstrates these interconnections. The kernel has the following responsibilities:

- Managing processes,
- Processing Client requests through a socket-based API, and
- Managing drivers.

Processes Processes allow multiple drivers to run simultaneously. Thus, concurrent installations can be performed. Processes are started, managed and removed through the API. The kernel must have three variables set for each process. Table 4 presents the required variables.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process ID</td>
<td>A unique integer identifying the process.</td>
</tr>
<tr>
<td>Driver</td>
<td>A driver must be bound to the process. The driver must be able to understand and communicate with the Target Node.</td>
</tr>
<tr>
<td>Interface</td>
<td>Operating system path to the Console-cable. Two processes can not share the same Console-cable, so this value must be unique.</td>
</tr>
</tbody>
</table>

It is possible to bind multiple processes with various drivers. Thus, multiplatform deployments can be performed simultaneously.

Process Management The kernel can both add and remove processes during operation. A process should only be removed when changing the active driver for a COM-port in the PIDS platform. A process does not have to be restarted between deployments, given that the active driver for the process can interpret all nodes. Given a scenario where an administrator must upgrade and configure four nodes:

- two Cisco Routers and
- two Juniper Routers.

The administrator runs a PIDS platform with two COM-ports (/dev/ttyS0, /dev/ttyS1). The administrator performs a concurrent deployment over the two ports. The administrator begins deploying the Cisco routers. In this example, the administrator needs one process per COM-port. The administrator creates two processes and binds them to their corresponding COM-port and driver. Table 5 shows the current process-list for the PIDS kernel. The deployment is initiated and finalized as shown in Figure 11.
Now, deployment is performed on the Juniper routers. The old processes can not be reutilized since they are bound to the Cisco-driver. Thus, the old processes must be removed and new processes must be added. Table 6 demonstrates the new process-list before deployment of the Juniper routers. The second deployment is initiated and finalized as shown in Figure 12.

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Driver</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>juniper</td>
<td>/dev/ttyS0</td>
</tr>
<tr>
<td>3</td>
<td>juniper</td>
<td>/dev/ttyS1</td>
</tr>
</tbody>
</table>

Table 6: PIDS kernel process list after modification.

Figure 12: Concurrent deployment of two Juniper routers.

3.3 API

The kernel is managed through a socket-based API, which separates the kernel from the GUI and allows for remote connections to the PIDS service. Because the API is openly documented, any engineer is able to develop their own implementation of a GUI for the service.

Communications over the API utilize a Request-Response method. The Client initiates a Request and the Server responds to the client with a status-packet (Response). The packet might include additional data if required. Depending on the Request-packet type, the client knows what additional variables to expect in the Response-packet. Table 7 presents the available Request packet types:
### Request Type

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCADD</td>
<td>Starts a new process.</td>
</tr>
<tr>
<td>PROCDATA</td>
<td>Removes an existing process.</td>
</tr>
<tr>
<td>PROCLIST</td>
<td>Lists the running processes.</td>
</tr>
<tr>
<td>SETUP</td>
<td>Prepares a node for deployment.</td>
</tr>
<tr>
<td>ANALYZE</td>
<td>Analyzes a node before deployment to examine available methods.</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>Initiates node deployment based on a predefined method.</td>
</tr>
</tbody>
</table>

Table 7: Request-packet types in PIDS API.

### Packet Format

TCP allows for both local and remote process-to-process communication. TCP is a reliable communications protocol [15, p.4]. A PIDS packet is encapsulated in the data field of a TCP packet. It consists of the following fields:

1. PIDS version field
2. Packet Type field
3. Variable fields (*Zero or More*)
4. Attribute fields (*Zero or More*)

All data is sent as an uncompressed string of text with fields separated with the ASCII character combination 0x0D0A (`\r\n`) or 0x0A (`\n`). Variables provide the kernel with additional information regarding the Request. Attributes provide subprocesses (like drivers) with additional information regarding the Request.

- **Variables** are identified as: `VAR_NAME=VAR_VALUE`
- **Attributes** are identified as: `ATTR_NAME:ATTR_VALUE`

Listing 1 demonstrates a PROCADD packet requesting to create a process through the PIDS API:

```
PIDS 1.0 \r\n
PROCADD \r\n
DRIVER=cisco \r\n
VTY:/dev/ttyS0
```

Listing 1: Example of a PROCADD request packet.

The Server responds the client with a status packet and a Process ID, as shown in Listing 2:

```
PIDS 1.0 \r\n
OK \r\n
ID: 0
```

Listing 2: Example of a PROCADD response packet.
3.4 User Interface

The only way to manage the PIDS kernel is through the API described in Section 3.3. The API can be communicated with directly via a standard telnet session or through a GUI. A HTML-based GUI is developed for demonstration purposes in this thesis. The GUI provides the following functionality:

- Process management:
  - Listing
  - Creation
  - Deletion
- Deployment management
- Troubleshooting errors through an interactive terminal

The device hosting the GUI-application is titled a PIDS Client according to the PIDS Architecture definitions. Figure 13 demonstrates a PIDS Client connecting to a PIDS Server. The same device hosting the kernel can also host the GUI. In that case, the GUI-application connects to the local loopback address: 127.0.0.1

Figure 13: Client connecting to a server through PIDS API.

Home The index page of PIDS GUI is called the Home view. It provides the administrator with direct links to PIDS documentation and other services of the platform as shown in Figure 14.

![Home view](image)

Figure 14: The Home-view in PIDS GUI.
Adding processes  In order to perform a deployment, a process must be added to the kernel. The process must be bound to a driver and a COM-port. Figure 15 demonstrates a scenario where a process is added, the process is bound to driver \texttt{cisco} and to the COM-port \texttt{/dev/ttyS0}.

In this particular scenario, the driver is not asking for additional data from the administrator. If the driver needs more data to work, it can request this data from the kernel. Additional HTML form inputs would be presented to the administrator through the GUI if that was the case.

![Figure 15: Adding processes in PIDS GUI.](image)

Managing processes  Processes can be managed through the Process view. The Process view performs the following functionality:

- Add processes.
- Remove running processes.
- Enter \texttt{Process Execution Mode} to perform deployments. Process Execution Mode is presented in the next paragraph.

![Figure 16: Viewing active processes in PIDS GUI.](image)
Process Execution Mode  The deployments are performed through Process Execution Mode. Figure 17 demonstrates this mode. Process Execution Mode consist of three elements:

- **Driver Methods** - The driver has a set of predefined methods. A method defines a certain function that the driver performs on the Target Node.

- **Driver Terminal** - The Driver Terminal displays the console cable output from the Target Node in real-time. If something goes wrong during a deployment, the issue can be identified through this terminal.

- **Driver Workspace** - The Driver Workspace informs the administrator regarding current deployment status and if additional data is requested by the driver. Figure 18 demonstrates a scenario where the driver needs additional data to perform a deployment.

Figure 17: Analyzing devices in PIDS GUI.

Figure 18: Selecting setup profile for devices in PIDS GUI.
3.5 Platform Drivers

Platform drivers control certain types of Target Nodes that are attached to the PIDS system. Drivers perform the actual node deployment through instructions received from the kernel. Drivers define the complete deployment procedure, including device access strategies and configuration methods. For example, when a certain device is only accessible through telnet, a driver must be available to allow for this device access strategy.

When an administrator is developing a deployment strategy for a Target Node, the administrator should never be supposed to interfere with the drivers source code. Instead, each driver has a simplified method for administrators to define deployment strategies (Profiles).

3.5.1 Profiles

Deployment strategies are defined within driver profiles. Drivers interpret profiles in real-time during deployments. It is up to the driver developer to define a profile syntax that is suitable for the specific driver.

Multiple profiles can be created for a single platform. The kernel must therefore inform the driver what profile to select before deployment. The kernel can instruct the driver to perform a device analysis and return a list of available profiles for a connected Target Node.

Standard Profile Syntax  A standard profile syntax (SPS) is defined for PIDS. It is up to the developer if they chose to implement this suggestion, but it is highly recommended in order to keep PIDS consistent and simplify administrative workload.

SPS is based on XML\(^\text{10}\) and provides the administrator with the option to input variable data prior to each deployment. As mentioned in Section 2, variable data inputs are especially necessary when deploying devices that require unique hostnames or IP addresses.

All SPS-based profiles are stored within a file named `profiles.xml`. The xml-root of the file consists of one driver tag for each driver connected to the PIDS kernel. Each driver-tag will in turn consist of zero or more profile-tags defining deployment profiles within the driver. Listing 3 demonstrates multiple driver-tags in a xml-root.

```
< CISCO >
  ...
</ CISCO >
< JUNIPER >
  ...
</ JUNIPER >
```

Listing 3: Example of driver-tags.

Profiles are defined with a tag named profile. The profile tag requires the following attributes:

- **id** - Unique name for the specific profile.
- **platform** - Identifies the specific platform for which the profile belongs.

The platform-attribute of a profile must match the platform of a Target Node for a deployment to start. Listing 4 demonstrates multiple profile-tags in a driver-tag.

```
< CISCO >
  < PROFILE id="Customer_A" platform="ME340x" >
    ...
  </ PROFILE >
  < PROFILE id="CPE_Baseline" platform="ME340x" >
    ...
  </ PROFILE >
</ CISCO >
```

Listing 4: Example of profile-tags within a driver named cisco.

\(^\text{10}\)April 2016 - [https://www.w3.org/TR/REC-xml/](https://www.w3.org/TR/REC-xml/)
A profile consists of four tags:

- `<image>` - Contains information of what system image the profile is compatible with. The image-tag both specifies what system image the Target Node is expected to run after the deployment and also where to find the correct system image if an upgrade is needed. Listing 5 demonstrates the image-tag. The location-tag informs the driver on the location of the system image if an upgrade is needed.

```
<profile id="Customer_A" platform="ME340x">
  <image version="12.2(60)EZ5">
    <location>http://172.16.0.1/me340x-122-60.EZ5.bin</location>
  </image>
</profile>
```

Listing 5: Example of an image-tag.

- `<variables>` - Defines zero or more variables an administrator is expected to provide prior to deployment. The list below demonstrates how to declare and reference variables:
  
  - Declaring: `<VAR_NAME description="INFO" />
  - Referencing: `%VAR_NAME%

In the example provided in Listing 6, the administrator is expected to provide PIDS with a hostname of the device before each deployment.

```
<profile id="Customer_A" platform="ME340x">
  <variables>
    <HOSTNAME description="Hostname of the Target Node" />
  </variables>
  <pre>
    <!-- Remove the current configuration. -->
    write erase
  </pre>
</profile>
```

Listing 6: Example of the variable-tag.

- `<pre>` - Defines a set of actions PIDS should perform on the Target Node prior to a system image upgrade, for instance setting the time or copying the current configuration to a remote fileserver.

- `<post>` - Defines a set of actions PIDS should perform on the Target Node after the system image upgrade, for instance setting a unique hostname, IP address and RSA Key-pair.

Actions are defined within the pre- and post-tags. Actions inform the driver of what to do and in what order it should be done. Listing 7 demonstrates how actions are called within profiles:

```
<profile id="Customer_A" platform="ME340x">
  <variables>
    <HOSTNAME description="Hostname of the Target Node" />
  </variables>
  <pre>
    <!-- Remove the current configuration. -->
    write erase
  </pre>
  <raw>
    clock set 12:00:00 JAN 1 2015
    configure terminal
    sdm prefer default
  </raw>
</profile>
```

Listing 7: Example of the pre- and post-actions.
interface Vlan 1
  ip address 172.16.0.2 255.255.255.0
  no shutdown

interface GigabitEthernet 0/1
  no shutdown

Listing 7: Example of actions called within the pre- and post-tag.

Actions are defined within the pre- and post-tags. Actions are driver-specific. Thus, actions available for the Juniper driver might differ or not work at all for the Cisco driver. The action `<write>` instructs PIDS to perform all steps necessary to save the current configuration. The action `<copy>` instructs PIDS to remotely or locally copy a file.

The `<raw>` action sends raw data over the console cable. This is practical in cases when there are no useful actions defined. Listing 8 demonstrates how a node can be configured by calling the `<raw>` action.

Listing 8: Example usage of the raw-tag.

```
<raw>
  configure terminal
  interface vlan 1
  ip address 192.168.0.1 255.255.255.0
  description This is a VLAN interface.
</raw>
```

3.5.2 Kernel Instructions

Drivers run independently and separated from the kernel. The kernel is responsible for managing drivers (processes) and to translate API request to driver instructions. This section explains how instructions are sent to drivers from the kernel.

PIDS is based on the object oriented language python. Drivers are classes based on a skeleton class defining what methods shall be callable by the kernel. The skeleton class provides a template when developing new drivers for PIDS. The kernel provides instructions to drivers based on the methods as defined by the skeleton class. Table 8 explains what methods must be callable for drivers in PIDS in order for the kernel to provide instructions.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Called when a driver is bound to a new process.</td>
</tr>
<tr>
<td>setup</td>
<td>Verify connectivity and prepare the connected device for deployment.</td>
</tr>
<tr>
<td>echo</td>
<td>Verify connectivity to the connected device.</td>
</tr>
<tr>
<td>analyze</td>
<td>Check what profiles are defined and available for the connected device.</td>
</tr>
<tr>
<td>execute</td>
<td>The execute method instructs the driver to execute a profile to the connected device.</td>
</tr>
<tr>
<td>stop</td>
<td>The stop method asks the driver to stop its operation.</td>
</tr>
</tbody>
</table>

Table 8: Drivers methods that must be callable by the kernel.
3.5.3 Console Cable Communication

According to the study presented in Section 2.3, most modern Customer-Premises Equipment must initially be configured through a console cable. For this reason, a standard toolset (PIDS Developer Tools) are developed to simplify console cable communications. The toolset is based on PySerial and is intended for drivers.

A console object (yost) is defined in the toolset. Several methods exist within the object controlling how information should be sent and read over the console cable. Table 9 presents the methods available in *yost*.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>init</em>()</td>
<td>Creates a console object and allocates the COM port specified in the port variable.</td>
</tr>
<tr>
<td>execute(line)</td>
<td>Sends the command included in the line-variable over the console cable. The method will return an integer identifying the specific command.</td>
</tr>
<tr>
<td>send(data)</td>
<td>Send raw data over the console cable.</td>
</tr>
<tr>
<td>listen()</td>
<td>Starts a thread listening for information sent over the console cable.</td>
</tr>
<tr>
<td>close()</td>
<td>Closes the thread listening for information sent over the console cable.</td>
</tr>
<tr>
<td>output(id)</td>
<td>Returns the collected output (as a list of strings) from the command specified with id.</td>
</tr>
</tbody>
</table>

Table 9: Methods available for the yost object.

Drivers can include the toolset by importing *tools* to the source code. Listing 9 demonstrates how the tool can simplify communication to a Cisco router over a console cable.

```python
import tools

# Creating a console object.
# COM port : /dev/ttyS0
# COM speed: 9600
vty = tools.yost("/dev/ttyS0", 9600)

# Enabling the listen thread.
vty.listen()
vty.execute("enable")

# Executing 'show ip interface brief'.
# The command identification is stored in variable cid.
cid = vty.execute("show ip interface brief")

# Printing the output generated from the command earlier.
for line in vty.output(cid):
    print(line)

# Closing the listen thread.
vty.close()
```

Listing 9: Console cable communication with PIDS Developer Tools.

3.6 Console Cable Execution Methods

There is no standardized way for the router to inform PIDS that an instruction has stopped processing. For example, when PIDS informs a router to generate SSH-keys with the command ```crypto key generate rsa modulus 2048``` , the router might process the command without providing any output for a long time. How does PIDS know that the router is ready to receive the next instruction? PIDS provides two execution-methods over console cable connections to solve this issue:

- Hold Based Execution Method
- Expect Based Execution Method

Hold Based Execution Method is the default method for PIDS because of its simplicity in operation.
Hold Based Execution Method

Hold Based Execution Method (HBEM) is a method where PIDS waits for a specified amount of time (hold time) before assuming that the device is ready for further instructions. The default hold time for PIDS is two seconds. If no data has been received from the Target Node within two seconds after executing a command, PIDS assumes that the Target Node is ready for further instructions. Two events reset the hold time:

- When instructions are sent to a Target Node.
- When data is received from a Target Node.

Figure 19 demonstrates how the hold timer operates during console cable communications given PIDS has two instructions (#1 and #2) in its instruction queue.

![Figure 19: Hold Timer operation in HBEM communication.](image)

The listen thread is responsible of reseting the hold timer when data is received from the Target Node. The execute method both reads and resets the timer as described below:

1. The execute method waits for the hold timer to allow for data transmission.
2. Data is transmitted.
3. The hold timer is reset with the default value (two seconds for PIDS).

The send method will neither wait for nor reset the hold-timer.

Expect Based Execution Method

Some instructions might result in varied output intervals where a two second hold-timer is not sufficient. Real world examples are:

- **RSA Key creation** - This might take up to a minute depending on key-size and system performance.
- **Remote and local file copying** - The transmission time depends on file size, system- and network performance.
- **Scripts** - Scripts might include sleep timers longer than two seconds.

```
1 echo "Sleeping for 10 seconds."
2 sleep 10
3 echo OK
```

Listing 10: script

Consider that PIDS instructs a Target Node to execute the script presented in Listing 10 with the Hold Based Execution Method. The command in Line 2 exceeds the hold timer above the standard two seconds. When the Hold Timer is exceeded, PIDS assumes the Target Node to be ready for further instructions. This assumption is wrong because the script will run for at least eight more seconds. This could be solved by increasing the Hold Timer to eleven seconds. Increasing the Hold Timer causes the following issues:
• Processes with output intervals of more than ten seconds will still cause issues.
• Processes with no significant output interval will experience performance loss since PIDS waits ten seconds before performing even the simplest instructions.

The Expect Based Execution Method (EBEM) resolves these issues. Instead of basing assumptions on Hold Timers, EBEM reads the output from a Target Node and waits for a predefined string (expect data) to appear. If the predefined string appears in the output, EBEM assumes the Target Node to be ready for next instruction. EBEM with expect data as OK would be preferable for the script demonstrated in Listing 10; this process is visualized in Figure 20.

![Diagram](image-url)

Figure 20: Demonstration of Expect Based Execution Method.
4 Analyzing PIDS

A couple of tests are conducted to determine at what speed PIDS performs deployments of Customer-Premises Equipment and how multiple processes and simultaneous deployments affect the performance and likelihood of errors. The following comparisons were performed:

- **Test 1** - Deployment of a single node:
  - A - Where an image upgrade is needed.
  - B - Where an image upgrade is not needed.

- **Test 2** - Simultaneous deployment of two nodes:
  - A - Where an image upgrade is needed.
  - B - Where an image upgrade is not needed.

- **Test 3** - Simultaneous deployment of three nodes:
  - A - Where an image upgrade is needed.
  - B - Where an image upgrade is not needed.

Each test is executed five times. If an error occurs, it is documented and the test is executed again.

4.1 Design

The equipment is set up as demonstrated in Figure 21. The PIDS platform is connected to a console cable hub bound to multiple processes to allow for simultaneous deployments.

The following Customer-Premises Equipment is selected for analysis:

- Cisco ME3400
- Cisco 800

The following list of criterias must be satisfied for a deployment to be considered successful:

1. Perform the following actions before image upgrade:
   
   (a) Backup the current configuration to a file server.
(b) Remove the current configuration.
(c) Set clock to 1 JAN 2016.
(d) Set sdm template to default.

2. Upgrade or downgrade the system to a version compatible with the template configuration (if needed).

3. Perform the following actions after image upgrade:
   (a) Set a device hostname.
   (b) Set a device domain name.
   (c) Generate RSA key-pairs.
   (d) Save the configuration.

A PIDS profile is developed for each platform. Listing 4.1 demonstrates the profile for the Cisco ME3400. XML-comments describe what criteria are to be fulfilled by the following command:

```xml
cisco
<profile id="Customer_A" platform="ME340x">
  <variables>
    <HOSTNAME description="Hostname of the Target Node" />
  </variables>

  <image version="12.2(60)EZ5">
    <location>http://172.16.0.1/me340x-122-60.EZ5.bin</location>
  </image>

  <!-- Actions performed before the image upgrade. -->
  <pre>
    <!-- Backup the current configuration to a file server. -->
    <copy src="nvram:startup-config" dst="tftp://172.16.0.1/%HOSTNAME%.old" />
    <!-- Remove the current configuration. -->
    <write erase />
    <!-- Set clock to 1 JAN 2016. -->
    <execute mode="privileged">clock set 12:00:00 JAN 1 2016</execute>
    <!-- Set sdm template to default. -->
    <execute mode="config">sdm prefer default</execute>
    <!-- Prepare the device for image upgrade. -->
    <raw>
      configure terminal
      interface Vlan 1
      ip address 172.16.0.2 255.255.255.0
      no shutdown
      interface GigabitEthernet 0/1
      no shutdown
    </raw>
  </pre>

  <!-- Actions performed after the image upgrade. -->
  <post>
    <!-- Download the template configuration from a file server. -->
    <copy src="http://172.16.0.1/switch_config" dst="nvram:startup-config" />
    <!-- Activate the configuration template. -->
    <configure_replace>nvram:startup-config</configure_replace>
    <!-- Set a device hostname. -->
    <execute mode="config">hostname %HOSTNAME%</execute>
    <!-- Set a device domain name. -->
    <execute mode="config">ip domain-name test</execute>
    <!-- Generate RSA key-pairs. -->
    <crypto_key_generate modulus="1024" />
    <!-- Save the configuration. -->
    <write />
  </post>
</profile>
```

28
4.2 Test 1: Single Deployment

This test measures the time it takes for PIDS to perform deployment of a single node.

**Method**  Figure 22 demonstrates the topology for this test.

![PIDS analysis - Topology for test 1](image)

A process is bound to the console cable in PIDS.

**Results**  Analysis 1A measures the time it takes to perform deployment on a single node when no image upgrade is required. Table 10 presents the result of this analysis.

<table>
<thead>
<tr>
<th>Device</th>
<th>OS upgrade?</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME340x</td>
<td>No</td>
<td>40 sec</td>
<td>45 sec</td>
<td>48 sec</td>
<td>40 sec</td>
<td>39 sec</td>
</tr>
<tr>
<td>C800</td>
<td>No</td>
<td>40 sec</td>
<td>40 sec</td>
<td>41 sec</td>
<td>40 sec</td>
<td>40 sec</td>
</tr>
</tbody>
</table>

Table 10: Test result of analysis 1A.

Analysis 1B measures the time it takes to perform deployment of a single node with an image upgrade. Table 11 present the result of this analysis.

<table>
<thead>
<tr>
<th>Device</th>
<th>OS upgrade?</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME340x</td>
<td>Yes</td>
<td>340 sec</td>
<td>341 sec</td>
<td>339 sec</td>
<td>339 sec</td>
<td>340 sec</td>
</tr>
<tr>
<td>C800</td>
<td>Yes</td>
<td>379 sec</td>
<td>380 sec</td>
<td>380 sec</td>
<td>382 sec</td>
<td>379 sec</td>
</tr>
</tbody>
</table>

Table 11: Test result of analysis 1B.

**Evaluation**  For this setup, the average deployment time was:

- ME340x
  - No upgrade: 42.4 seconds
  - With upgrade: 5 minutes and 39.8 seconds
- C800
  - No upgrade: 40.2 seconds
  - With upgrade: 6 minutes and 20 seconds
4.3 Test 2: Dual Deployment

This test measures the time it takes for PIDS to perform simultaneous deployment of two nodes.

**Method** Figure 23 demonstrates the topology for this test.

![Figure 23: PIDS analysis - Topology for test 2.](image)

There are in total two processes managing the console ports in the topology.

**Results** Analysis 2A measures the time it takes to perform simultaneous deployment of two nodes when no image upgrade is required. Table 12 presents the result of this analysis.

<table>
<thead>
<tr>
<th>Device</th>
<th>OS upgrade?</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME340x</td>
<td>No</td>
<td>40 sec</td>
<td>40 sec</td>
<td>43 sec</td>
<td>45 sec</td>
<td>42 sec</td>
</tr>
<tr>
<td>C800</td>
<td>No</td>
<td>35 sec</td>
<td>41 sec</td>
<td>40 sec</td>
<td>41 sec</td>
<td>40 sec</td>
</tr>
</tbody>
</table>

Table 12: Test result of analysis 2A.

Analysis 2B measures the time it takes to perform simultaneous deployment of two nodes where an image upgrade is performed on both nodes. Table 13 presents the result of this analysis.

<table>
<thead>
<tr>
<th>Device</th>
<th>OS upgrade?</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME340x</td>
<td>Yes</td>
<td>347 sec</td>
<td>349 sec</td>
<td>346 sec</td>
<td>343 sec</td>
<td>348 sec</td>
</tr>
<tr>
<td>C800</td>
<td>Yes</td>
<td>384 sec</td>
<td>384 sec</td>
<td>382 sec</td>
<td>383 sec</td>
<td>381 sec</td>
</tr>
</tbody>
</table>

Table 13: Test result of analysis 2B.

**Evaluation** For this setup, the average deployment time was:

- ME340x
  - **No upgrade:** 42 seconds
  - **With upgrade:** 5 minutes and 46.6 seconds

- C800
  - **No upgrade:** 39.4 seconds
  - **With upgrade:** 6 minutes and 22.8 seconds
4.4 Test 3: Triple Deployment

This test measures the time it takes for PIDS to perform simultaneous deployment of three nodes.

**Method** A third device (Cisco C181X) are selected to participate in the final analysis. Figure 24 demonstrates the topology for this test.

![Topology for Test 3](image)

There are in total three processes managing the console ports in the topology.

**Results** Analysis 3A measures the time it takes to perform simultaneous deployment of three nodes when no image upgrade is required. Table 14 presents the result of this analysis.

<table>
<thead>
<tr>
<th>Device</th>
<th>OS upgrade?</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME340x</td>
<td>No</td>
<td>40 sec</td>
<td>45 sec</td>
<td>42 sec</td>
<td>40 sec</td>
<td>41 sec</td>
</tr>
<tr>
<td>C800</td>
<td>No</td>
<td>40 sec</td>
<td>40 sec</td>
<td>41 sec</td>
<td>41 sec</td>
<td>40 sec</td>
</tr>
<tr>
<td>C181X</td>
<td>No</td>
<td>51 sec</td>
<td>50 sec</td>
<td>51 sec</td>
<td>52 sec</td>
<td>50 sec</td>
</tr>
</tbody>
</table>

Table 14: Test result of analysis 3A.

Analysis 3B measures the time it takes to perform simultaneous deployment of three nodes where a image upgrade is performed on all nodes. Table 15 presents the result of this analysis.

<table>
<thead>
<tr>
<th>Device</th>
<th>OS upgrade?</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME340x</td>
<td>Yes</td>
<td>397 sec</td>
<td>400 sec</td>
<td>400 sec</td>
<td>398 sec</td>
<td>397 sec</td>
</tr>
<tr>
<td>C800</td>
<td>Yes</td>
<td>409 sec</td>
<td>409 sec</td>
<td>408 sec</td>
<td>410 sec</td>
<td>408 sec</td>
</tr>
<tr>
<td>C181X</td>
<td>Yes</td>
<td>361 sec</td>
<td>364 sec</td>
<td>362 sec</td>
<td>362 sec</td>
<td>359 sec</td>
</tr>
</tbody>
</table>

Table 15: Test result of analysis 3B.

**Evaluation** For this setup, the average deployment time was:

- **ME340x**
  - No upgrade: 41.6 seconds
  - With upgrade: 6 minutes and 38.4 seconds
- **C800**
  - No upgrade: 40.4 seconds
  - With upgrade: 6 minutes and 48.8 seconds
5 Interpretation

5.1 PIDS Performance

Deploying multiple nodes simultaneously does not significantly decrease the time required for node configuration. The performance slightly decreases only when multiple nodes require an image upgrade at the same time. This is very likely due to the increased load of the network.

Figure 25 demonstrates how the deployment time is affected (in seconds) by adding additional nodes where none of the nodes required a system image upgrade:

![Figure 25: Average deployment time (no system image upgrades).](image)

Figure 26 demonstrates how the deployment time is affected when deploying multiple devices requiring an image upgrade. The figure clearly shows the drop of performance when deploying three devices simultaneously, as compared to only two devices. As mentioned earlier, this drop of performance is very likely due to the increased network load from nodes downloading their corresponding system image from the remote file server.

![Figure 26: Average deployment time (with system image upgrades).](image)
5.2 PIDS versus Zero Touch Deployment Methods

In Section 2.2.2, two proprietary tools are presented for automated deployment. These tools are Cisco Smart Install and Juniper Zero Touch Provisioning. How do these tools differ from PIDS?

Table 16 compares general characteristics of PIDS, Cisco Smart Install (CSI) and Juniper Zero Touch Provisioning (JZTP).

<table>
<thead>
<tr>
<th>Feature</th>
<th>PIDS</th>
<th>CSI</th>
<th>JZTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>The software can be 100% automated.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Can run as a service on existing production networking equipment.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Can perform live deployments in production networks.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Can apply unique configurations for individual devices.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Can perform simultaneous deployments.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Software is based on In-band management protocols.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Software can perform deployments to equipment from multiple vendors.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>The software can perform deployment of routers.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Allow to enter parameters prior to each deployment.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Software requires manual intervention prior to each deployment.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 16: Comparison of PIDS to Zero Touch Deployment Methods.

PIDS performs deployments before the equipment is connected to a production network. The benefits of this are:

- Greater flexibility when configuring of the equipment.
- Increased administrator control over the specific device.

The person performing a deployment with PIDS only needs to know how to insert a console cable to the Target Node. PIDS performs the actual configuration and image upgrade if needed.

Deploying equipment with PIDS is generally a two steps process. First, the administrator connects PIDS to the Target Node and performs the upgrade and configuration. Then, the administrator connects the Target Node to the production network. According to Jörgen Jonasson (TDC), this process is often performed by electricians with limited knowledge about routers. With PIDS, these electricians will not need the additional education required to perform configurations for various platforms. Figure 27 demonstrates this process:

![Figure 27: PIDS two step deployment procedure.](image-url)
Zero Touch Deployment Methods perform deployment of switches after the equipment is connected to a production network. This is suitable for cases when the equipment has nearly identical configurations and must be replaced with little effort.

Figure 28 shows that Cisco Smart Install is configured for the switch block. The access switch S1 is experiencing problems and end users connected to the switch are reporting connectivity issues and packet loss.

The switch S1_new is of the same model as S1 and is chosen as a replacement for the device. Because the switchblock is configured to run Cisco Smart Install, the replacement switch needs only to be connected to the network as shown in Figure 29. Cisco Smart Install automatically upgrades and configures the device.

Figure 28: S1 experiences failure and must be replaced.

Figure 29: S1_new replaces S1 and is automatically configured by Cisco Smart Install.
6 Conclusion

The first objective was to determine state-of-the-art when deploying Customer-Premises Equipment. Chapter 2 provides the relevant discussion for this issue. No tool was found performing automated deployments of routers. This leads to Research Objective 2:

Develop a tool to perform an automated deployment of Customer-Premises Equipment (PIDS). PIDS is described in Chapter 3, analyzed in Chapter 4 and compared to existing Zero Touch Deployment Methods in Chapter 5.

Outlook

Section 5.2 explains that PIDS requires manual intervention prior to each deployment as opposed to current Zero Touch Deployment Methods. It should be further investigated if the application can be improved so that the requirement of manual intervention only is optional when deploying multiple devices.

The Standard Profile Syntax as explained in Section 3.5.1 can be further improved by defining methods based on conditions. A pseudo-code example can be If GigabitEthernet 0 exist Then ...

Adding drivers to PIDS requires slight modification of the kernel source code. Separating this process potentially simplifies driver management and development. An external file can for example store all driver information and be read by the kernel during startup.

Sanity checks are not fully implemented for PIDS as of publication of this thesis. Unexpected crashes occur when the software is treated improperly.
References


## Initial Configuration Study Manuals

Table 17 list all the documents used when reviewing the devices in the *Manual Node Deployment Study* presented in section 2.3.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Series</th>
<th>Manual Link</th>
</tr>
</thead>
</table>

Table 17: Manuals for the devices in the Initial Configuration Study.
Appendix B - PIDS Source Code Reference

Christian Åberg

1. KERNEL.PY
The PIDS Kernel manages the API communication, the processes and the drivers. The kernel object is controlled through PIDS API.

1.1 Methods
1.1.1 Public.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>init</em>()</td>
<td>Initiates kernel variables.</td>
</tr>
<tr>
<td>newCon(socket)</td>
<td>Notifies the kernel about an incoming socket. The kernel will create a separate thread to handle the communication.</td>
</tr>
</tbody>
</table>

1.1.2 Private.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conThread(socket)</td>
<td>Handles socket communication. The method interprets API requests and:</td>
</tr>
<tr>
<td></td>
<td>---performs kernel actions (like creating or removing processes) or</td>
</tr>
<tr>
<td></td>
<td>---translates the requests to driver instructions.</td>
</tr>
<tr>
<td>getProcById(identity)</td>
<td>Accepts an integer identification and returns the corresponding process from the process list array.</td>
</tr>
</tbody>
</table>

1.2 Source

```python
import cisco
import time
import socket
import threading

def getVar (orig_list, var_name):
    for line in orig_list:
        var = line.split("=")
        if len(var) == 2:
            if var[0] == var_name:
                return var[1]
```

PIDS Source Code Reference, Vol. 1, No. 1, Article 1, Publication date: May 2016
return None
class kernel:
    def __init__(self):
        self.next_proc_id = 0
        self.proc_list = []
        self.threads = []
    def getProcById(self, identity):
        if identity:
            for proc in self.proc_list:
                if proc[1] == int(identity):
                    return proc
        return False
    def conThread(self, socket):
        buffer = []
        while True:
            # API socket read.
            data = socket.recv(1024).decode("utf-8")

            if 2 < len(data.split("\r\n")):
                buffer = data.split("\r\n")
                break

            data = data.replace("\r", "")
            data = data.replace("\n", "")

            if not data:  # If client sent empty data. Break the loop.
                break

            buffer.append(data)

            if 2 < len(buffer):
                if buffer[0] == "PIDS_0.1":  
                    if buffer[1] == "SETUP": 
                        proc = self.getProcById(getVar(buffer, "ID")) 
                        if proc:
                            proc[0].setup() 
                            socket.send(b"PIDS_0.1\n") 
                            socket.send(b"OK\n")
                if buffer[1] == "PROCADD": 
                    driver = getVar(buffer, "DRIVER") 
                    vty = getVar(buffer, "VTY")

                    if driver == "cisco" and vty:
                        self.proc_list.append([cisco.driver(vty, 9600), self.next_proc_id, "cisco", vty])

                        socket.send(b"PIDS_0.1\n")
                        socket.send(b"OK\n")
socket.send("ID:")+str(self.next_proc_id) + ".encode()
self.next_proc_id += 1

else:
    socket.send(b"PIDS_0.1\n")
    socket.send(b"FAIL\n")
    socket.send(b"REASON=Missing+arguments.\n")

elif buffer[1] == "PROCDEL":
    proc = self.getProcById(getVar(buffer, "ID"))
    if proc:
        if proc[0].stop():
            self.proc_list.remove(proc)
            socket.send(b"PIDS_0.1\n")
            socket.send(b"OK\n")
        else:
            socket.send(b"PIDS_0.1\n")
            socket.send(b"FAIL\n")
            socket.send(b"REASON=Process+is+busy!\n")
    elif buffer[1] == "ECHO":
        proc = self.getProcById(getVar(buffer, "ID"))
        if proc:
            if proc[0].echo():
                socket.send(b"PIDS_0.1\n")
                socket.send(b"OK\n")
            else:
                socket.send(b"PIDS_0.1\n")
                socket.send(b"FAIL\n")

elif buffer[1] == "PROCLIST":
    socket.send(b"OK\n")
    for process in self.proc_list:

elif buffer[1] == "ANALYZE":
    proc = self.getProcById(getVar(buffer, "ID"))
    if proc:
        report = proc[0].analyze()
        if report:
            socket.send(b"PIDS_0.1\n")
            socket.send(b"OK\n")
            for line in report:
                socket.send((line + ".\n") .encode())
        else:
            socket.send(b"PIDS_0.1\n")
            socket.send(b"FAIL\n")
            socket.send(b"REASON=Analyze+did+not+start+properly.\n")

elif buffer[1] == "EXECUTE":
    identity = getVar(buffer, "ID")
profile = getVar(buffer, "PROFILE")

if identity and profile:
    profile_variables = []
    for line in buffer:
        var = line.split("":"")

        if len(var) == 2:
            profile_variables.append([var[0], var[1]])

    for proc in self.proc_list:
        if proc[1] == int(identity):
            socket.send(b"PIDS_0.1\n")
            socket.send(b"OK\n")
        elif buffer[1] == "READLINE":
            identity = getVar(buffer, "ID")
            line_pos = getVar(buffer, "LINE")

            if identity and line_pos:
                proc = self.getProcByld(identity)
                if proc:
                    socket.send(b"PIDS_0.1\n")
                    socket.send(b"LINEBULK")
                    lines = proc[0].readline(int(line_pos))
                    for line in lines:
                        socket.send("\n" + line).encode()

        socket.close()

def newCon(self, socket):
    t = threading.Thread(target=self.conThread, args=(socket,))
    self.threads.append(t)
    t.start()

srvSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srvSocket.bind(("127.0.0.1", 2000))
print("Starts to listen.")
srvSocket.listen(25)

engine = kernel()
while True:
    socket, addr = srvSocket.accept()
    engine.newCon(socket)
2. CISCO.PY

The driver object is the actual driver and is managed by the PIDS kernel. Drivers are bound to processes and receives instructions from the kernel through public methods defined as Skeleton Class Methods (SCM). Non SCM are considered private and will not be called by the kernel; these methods are defined as Private Driver Methods (PDM).

2.1 Methods

2.1.1 Skeleton Class Methods. The kernel translates API requests to corresponding SCM. Only the kernel calls SCM. The following SCM exist:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(port, speed)</td>
<td>Initiates driver variables.</td>
</tr>
<tr>
<td>2.1.1.1 Parameters</td>
<td></td>
</tr>
<tr>
<td>---port</td>
<td>Operating system path to the serial device. E.g: &quot;/dev/ttyS0&quot;</td>
</tr>
<tr>
<td>---speed</td>
<td>Bps of the serial device. E.g: 9600</td>
</tr>
<tr>
<td>setup()</td>
<td>Prepares the Target Node for additional instructions and verifies connectivity.</td>
</tr>
<tr>
<td>echo()</td>
<td>Verifies connectivity to the Target Node.</td>
</tr>
<tr>
<td>2.1.1.2 Return. True if verification succeeded, False otherwise.</td>
<td></td>
</tr>
<tr>
<td>execute(profile, attr)</td>
<td>Executes a deployment profile on the Target Node.</td>
</tr>
<tr>
<td>2.1.1.3 Parameters</td>
<td></td>
</tr>
<tr>
<td>---profile</td>
<td>Name of the deployment profile.</td>
</tr>
<tr>
<td>---attr</td>
<td>Additional variables to be transmitted to the deployment profile. E.g: [[&quot;VAR1&quot;,&quot;DATA&quot;],[&quot;VAR2&quot;,&quot;DATA&quot;]]</td>
</tr>
</tbody>
</table>
analyze() Performs an analysis of the Target Node. Target Node information and available deployment profiles (with its corresponding variables) are returned to the kernel.

2.1.1.4 Return. List of strings if analysis succeeded, None otherwise. E.g:
"["ATTR:DATA", "PROFILE=VAR,DSC;"]"

Where:
---ATTR - Attribute defining a property for a Target Node.
---DATA - Data for the attribute.
---PROFILE - Available deployment profile for the Target Node.
---VAR - Profile variable.
---DSC - Description of the variable.

Multiple attributes, profiles and variables can exist.

readline(pos) Return collected Target Node output as a list of strings from a specific position in the buffer.

2.1.1.5 Parameters
---pos - Buffer position.

2.1.1.6 Return. List of strings. E.g.
"["NEXTLINE=4", "First line..", "Second..", "."]"

Where:
---NEXTLINE - Lowest empty buffer position. Next time buffer receives input, the input is stored in this position.

stop() Stops driver operation.

2.1.1.7 Return. False if deployments are active, otherwise True and driver stops.

2.1.2 Private Driver Methods. PDM are called within the driver but do not have to be known to the kernel. SCM calls PDM. The following PDM exist:

vf(data) Reads input string and replaces profile variables (%VARNAME%) with its corresponding value.

2.1.2.1 Return. String with variable names replaced to the actual variable value.

exeBlock(block, profile) The exeBlock method is part of the deployment profile interpretation mechanism. The exeBlock interprets pre and post tags and performs the actions defined within those tags.

2.1.2.2 Example. Execute the actions within the pre-tag: exeBlock("pre", profile)
deploy() The deploy method initiates Target Node deployments. This method is run as a separate thread. During a deployment, the driver can not be closed with the stop method.

2.2 Source

```python
import cisco_yost
import os
import time
import xml.etree.ElementTree as ET
import threading
class driver:
    def __init__(self, port, speed):
        self.xml_profile = None
        self.profile_node = None
        self.variables = []
        self.busy = False

        self.con = cisco_yost.cisco_yost(port, speed)

    def vf(self, data):
        data_fix = data
        for variable in self.variables:
            data_fix = data_fix.replace("%" + variable[0] + "%", variable[1])

        return data_fix

    def setup(self):
        if not self.con.wakeup():
            panic("Node is not responding.")

        info("Disabling console logging.")
        self.con.loggingDisable()

        tree = ET.parse('node−setup.xml')
        self.root = tree.getroot()

    def echo(self):
        if self.con.wakeup():
            return True
        else:
            return False

    def executeBlock(self, block, profile):
        if not self.con.wakeup():
```
panic("Node is not responding.")

for child in profile:
    if child.tag == block:
        for action in child:
            if action.tag == "execute":
                if action.get("mode") == "config" or action.get("mode") == "configuration":
                    self.con.execute("configure terminal")

            if action.get("expect") == None:
                self.con.execute(self.vf(action.text))
            else:
                self.con.expect(self.vf(action.text), [action.get("expect")])

            if action.get("mode") == "config" or action.get("mode") == "configuration":
                self.con.execute("end")
            elif action.tag == "copy":
                self.con.copy(self.vf(action.get("src")), self.vf(action.get("dst")))
            elif action.tag == "reload":
                self.con.reboot()
            elif action.tag == "configure_replace":
                self.con.configureReplace(action.text)
            elif action.tag == "raw":
                for bulk_action in action.text.splitlines():
                    self.con.vty.execute(self.vf(bulk_action).strip())
                self.con.execute("end")
            elif action.tag == "sleep":
                info("Sleeping " + action.text + " seconds!")
                time.sleep(int(action.text))
            elif action.tag == "write":
                self.con.expect("write", ["[OK]"])  
            elif action.tag == "write_erase":
                self.con.writeErase()
            elif action.tag == "crypto_key_generate":
                self.con.rsaGenerate(self.vf(action.get("modulus")))

def deploy(self):
    self.busy = True
    self.executeBlock("pre", self.profile_node)

    for child in self.profile_node:
        if child.tag == "image":

            image_id = child.get("image-id")
            software_version = child.get("software-version")

            for subchild in child:
if subchild.tag == "location":
    image_location = subchild.text
elif subchild.tag == "filename":
    image_filename = subchild.text

if not (self.con.image_id == image_id and self.con.software_version == software_version):
time.sleep(5)

info("Removing current system image.")
self.con.imageDelete()

info("Starting file download.")
if not self.con.copy(image_location + image_filename, "flash:"):
    panic("Could not download new system image from server.")

self.con.imageActivate("flash:" + image_filename)
self.con.reboot()

time.sleep(5)
self.con.wakeup()

self.executeBlock("post", self.profile_node)
self.busy = False

def execute(self, profile, attr):
    if not self.con.wakeup():
        panic("Node is not responding.")

    self.con.identify()

    profile_found = False

    for profile_candidate in self.root:
        if profile == profile_candidate.get("id") and not profile_found:
            profile_found = True
            self.profile_node = profile_candidate
            self.variables = attr
            self.execution_thread = threading.Thread(target=self.deploy)
            self.execution_thread.start()

        if not profile_found:
            print("No profile could be found!")

def analyze(self):
    if not self.con.wakeup():
        panic("Node is not responding.")
return None

self.con.identify()

output = []
output.append("Platform:" + self.con.platform)
output.append("Image-id:" + self.con.image_id)
output.append("Software-version:" + self.con.software_version)
output.append("Software_type:" + self.con.software_type)

for child in self.root:
    if self.con.platform == child.get("platform"):
        tmp = child.get("id") + "="
        for subchild in child:
            if subchild.tag == "variables":
                for variable in subchild:
                    tmp += variable.tag + "," + str(variable.get("description")) + ";"
        output.append(tmp)
return output

def readline(self, pos):
    output = []
    while pos < len(self.con.vty.buffer):
        output.append(self.con.vty.buffer[pos])
        pos += 1
    output.insert(0, "NEXTLINE=" + str(pos))
    return output

def stop(self):
    if not self.buzy:
        self.con.vty.close()
        return True
    else:
        return False

def info(string):
    print("INFO:" + string)

def panic(string):
    print("ERROR:" + string)
od.os._exit(1)
3. CISCO_YOST.PY

The cisco_yost object defines a set of methods that either automates or simplifies communication with cisco devices over console cables. The object utilizes the yost class that is included in PIDS Developer Tools. Methods within the cisco_yost object are defined as either:

—Yost Communication Methods
—Profile Interpreter Methods

The cisco_yost object is initiated with:

```
_init_(port, speed)
```

Initiates cisco_yost variables.

3.0.0.3 Parameters

---port - Operating system path to the serial device. E.g: 
"/dev/ttyS0"

---speed - Bps of the serial device. E.g: 9600

3.1 Yost Communication Methods

Data is transferred to Cisco nodes over console cables with Yost Communication Methods (YCM). There are two YCM defined for the cisco_yost object:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute(command)</td>
<td>Hold Based Execution Method</td>
</tr>
<tr>
<td>expect(command, [GOOD], [BAD])</td>
<td>Expect Based Execution Method</td>
</tr>
</tbody>
</table>

3.1.0.4 Parameters

---command - Command to execute on the Cisco node (Hold Based Execution Method).

3.1.0.5 Return. Target Node response as a list of strings.

3.1.0.6 Parameters

---command - Command to execute on the Cisco node (Expect Based Execution Method).

---GOOD - List of strings containing data that is considered good.

---BAD - List of strings containing data that is considered bad.

3.1.0.7 Return. True if good data is matched, False if bad data is matched.

3.2 Profile Interpreter Methods

Some configuration or verification steps for Cisco nodes are completely automated through Profile Interpreter Methods. These methods are callable inside profiles. The following Profile Interpreter Methods are defined for the cisco_yost object:
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>identify()</td>
<td>Instructs the driver to identify the Target Node. If identification is successful, node information is stored in the following variables:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>---<code>cisco_yost.platform</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>---<code>cisco_yost.image_id</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>---<code>cisco_yost.software_version</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>---<code>cisco_yost.software_type</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>---<code>cisco_yost.active_image</code></td>
<td></td>
</tr>
<tr>
<td>wakeup()</td>
<td>Prepare the Target Node for further instructions and verify connectivity.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.2.0.8 Return. True if connectivity verification succeed; False is returned otherwise.</td>
<td></td>
</tr>
<tr>
<td>writeErase()</td>
<td>Delete the current Startup Configuration for the Target Node.</td>
<td></td>
</tr>
<tr>
<td>testICMP(ip)</td>
<td>Verify IP connectivity between the Target Node and the destination IP address (ip). The IP verification is based on ICMP.</td>
<td>---ip - Destination IP address.</td>
</tr>
<tr>
<td>loggingDisable()</td>
<td>Disable console logging on the Target Node. Console logging can have bad impact on PIDS performance and should therefore always be disabled.</td>
<td></td>
</tr>
<tr>
<td>imageActivate(image)</td>
<td>INFO: Configure a new active system image for the Target Node.</td>
<td>---image - Local path to the system image.</td>
</tr>
<tr>
<td>imageDelete()</td>
<td>Delete the current active image for the Target Node. This is preferably performed before downloading a new image to preserve filesystem space.</td>
<td></td>
</tr>
<tr>
<td>copy(src, dst)</td>
<td>Perform a copy on the Target Node.</td>
<td>---src - Copy source. Cisco URL syntax E.g: flash:system-image or <a href="http://10.0.0.1/system-image">http://10.0.0.1/system-image</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>---dst - Copy destination. Cisco URL syntax.</td>
</tr>
<tr>
<td>reboot()</td>
<td>Instructs the driver to reboot the Target Node.</td>
<td></td>
</tr>
</tbody>
</table>
rsaGenerate(modulus) Generate Rivest, Shamir and Adelman (RSA) key pairs for the Target Node.

3.2.0.13 Parameters
---modulus - Size of the key modulus.

configureReplace(src) Replace configuration for the Target Node.

3.2.0.14 Parameters
---src - Path to the configuration file.

close() Instructs the driver to close its operation and running threads.

3.3 Source

```python
import tools
import time
class cisco_yost:
    def __init__(self, path, speed):
        self.vty = tools.yost(path, speed)
        self.vty.listen()

        self.platform = ""
        self.image_id = ""
        self.software_version = ""
        self.software_type = ""
        self.active_image = ""

    def execute(self, command):
        return self.vty.output(self.vty.execute(command))

    def expect(self, command, expect_list, error_list = []):
        check_position = 0

        output_id = self.vty.execute(command)

        while True:
            while check_position < len(self.vty.output(output_id)) - 1:
                check_position += 1

            for expect in expect_list:
                if expect in self.vty.output(output_id)[check_position]:
                    return True

            for error in error_list:
                if error in self.vty.output(output_id)[check_position]:
                    return False
```

PIDS Source Code Reference, Vol. 1, No. 1, Article 1, Publication date: May 2016.
### PROFILE INTERPRETER METHODS ###

```python
def identify(self):
    version = self.execute("show version\r\n")
    identification = version[1].split(",")
    tmp = identification[1]
    tmp = tmp.split("Software")
    self.platform = tmp[0]
    self.image_id = tmp[1][1:-1]
    self.software_version = identification[2].replace("Version", "")
    self.software_type = identification[3]

    for line in version:
        if "System image file is" in line:
            self.active_image = line.split(""")[1]

def wakeup(self):
    for i in range(25):
        self.execute("\x03")
        rec_buffer = self.execute("echo WAKEUP")
        if len(rec_buffer) > 1:
            if rec_buffer[1] == "WAKEUP" or "The command you have entered" in rec_buffer[1]:
                self.execute("enable")
                return True
            else:
                print("Insufficient lines received during wakeup.")
                return False

def writeErase(self):
    self.execute("write erase")
    return False

def testICMP(self, ip):
    for i in range(3):
        if self.expect("ping -c 3", ["Success", "rate is "]):
            return True
    return False

def loggingDisable(self):
    self.execute("configure terminal")
    self.execute("no logging console")
    self.execute("end")
```

PIDS Source Code Reference, Vol. 1, No. 1, Article 1, Publication date: May 2016.
def imageActivate(self, image):
    self.execute("configure terminal")
    self.execute("boot system" + image)
    self.execute("end")

def imageDelete(self):
    self.execute("delete" + self.active_image + "\r\r\r\r")

def copy(self, src, dst):
    if self.expect("copy" + src + " " + dst + "\r\r\r\r", ["bytes copied in"],
                  ["No such file or directory", "Timed out", "I/O error"]):
        return True
    else:
        return False

def reboot(self):
    self.execute("reload\rno\r", ["Press RETURN to get started!",
                                 "−−− System Configuration Dialog −−−"])
    self.execute("\r")
    self.execute("no")
    self.execute("enable")

def rsaGenerate(self, modulus):
    self.execute("configure terminal")

    self.expect("crypto key generate rsa modulus" + modulus, ["[OK]"],
                  ["The input file is not a valid config file")
    self.execute("end")

def configureReplace(self, src):
    return self.expect("configure replace" + src + "\rY", ["Rollback Done"],
                        ["The input file is not a valid config file")

def close(self):
    self.vty.close()
4. TOOLS.PY

The yost object is one of the tools included in the toolset PIDS Developer Tools. The object is built on
PySerial and has been adapted to simplify console cable communications with network nodes.

The listenThread() records responses received over the console cable. The responses are stored in
object variable yost.buffer. The listenThread() must be manually started. The method listen() starts
the listenThread().

Each time a command is executed with the execute(line) method, a command identification integer
is returned. The output(id) method returns a list of strings containing the response received from
command identified as 'id'.

4.1 Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>init</em>(port, speed)</td>
<td>Sets up the console cable.</td>
</tr>
<tr>
<td>listen()</td>
<td>Starts listenThread(). Each line that is read by listenThread() is appended to object variable yost.buffer.</td>
</tr>
<tr>
<td>close()</td>
<td>Closes the listen thread. The thread will not close until no data is expected.</td>
</tr>
<tr>
<td>execute(line)</td>
<td>Transmits data to a Target Node.</td>
</tr>
<tr>
<td>output(id)</td>
<td>Return a list of strings containing the response from a Target Node.</td>
</tr>
<tr>
<td>hold()</td>
<td>Resets the hold timer (Hold Based Execution Method). Default hold timer for PIDS is two seconds.</td>
</tr>
<tr>
<td>isReady()</td>
<td>Waits for the hold timer to be exceeded.</td>
</tr>
</tbody>
</table>

4.1.0.15 Parameters
---port - Operating system path to the serial device. E.g: 
"/dev/ttyS0"
---speed - Bps of the serial device. E.g: 9600

4.1.0.16 Parameters
---line - String containing the command that shall be executed over the console cable.

4.1.0.17 Return. Command identification integer.

4.1.0.18 Parameters
---id - Command identification integer for which the response shall be returned.

4.1.0.19 Return. List of strings.

4.2 Source

```
import serial
import threading
import time
import calendar
```

PIDS Source Code Reference, Vol. 1, No. 1, Article 1, Publication date: May 2016.
class yost():
def __init__(self, port, speed):
    self.con = serial.Serial(port, baudrate=speed)
    self.thread_object = threading.Thread(target=self.listen_thread)
    self.listening = False
    self.buffer = []
    self.command_pos = []
    self.hold_time = 0
def output(self, id):
    self.isReady()
    output_string = []
    start = self.command_pos[id]

    if id == len(self.command_pos) - 1:
        end = len(self.buffer)
    else:
        end = self.command_pos[id + 1]

    for i in range(start, end):
        output_string.append(self.buffer[i])

    return output_string
def listen(self):
    self.listening = True
    self.thread_object.start()
def listen_thread(self):
    while self.listening:
        line = self.con.readline().decode("utf8").rstrip()
        self.buffer.append(line)
        self.hold()
    def close(self):
        self.isReady()
    self.listening = False
    self.con.write(b'\x03')
def execute(self, line):
    self.isReady()
    self.command_pos.append(len(self.buffer))
    self.con.write((line + '\r').encode())
    self.hold()
    return len(self.command_pos) - 1
def hold(self):
    self.hold_time = calendar.timegm(time.gmtime()) + 2
def send(self, data):
    self.con.write(data)

def isReady(self):
    while calendar.timegm(time.gmtime()) < self.hold_time:
        time.sleep(1)