Provisioning Guide

On-Ramp Wireless, Inc.
10920 Via Frontera, Suite 200
San Diego, CA 92127
U.S.A.

Provisioning Guide

010-0074-00 Rev. C

April 16, 2014

Contents

1 Introduction 1

2 Provisioning Overview 1

2.1 Provisioning Server 1
2.2 Provisioning Client 1
2.3 Packaging Server 2

3 Key Management Overview 4

3.1 Key Generation and Key Management 4
3.2 Node Key Provisioning 4
3.3 KMS Key Generation and Import Node Keys Process 6

4 System Requirements 7

4.1 Provisioning Server 7
4.2 Provisioning Client 7
4.2.1 Windows-based Platforms 7
4.2.2 Linux-based Platforms 8
4.3 Shared File Repository 8

5 Installing and Configuring a Provisioning Server 9

5.1 Prerequisite Packages for the Provisioning Server Installation 9
5.1.1 Java 9
5.1.2 MySQL Server 10
5.1.3 Java Cryptography Extension 10
5.2 Installing the Provisioning Server 11
5.3 Configuring the Database 14

6 Installing and Configuring a Provisioning Client 16

6.1 Installation on a Windows-based Client 16
6.2 Installation on a Linux-based Client 16
6.2.1 Installing PyCrypto on a CentOS 6.x/RHEL 6.x System 17
6.2.2 Installing PySerial on a CentOS 6x/RHEL 6.x System 17
6.2.3 Installing PyCrypto and PySerial on an Ubuntu/Debian System 17
6.2.4 Installing the Provisioning Client 18

7 Operation 19

7.1 Adding or Modifying a Device or Network 19

7.1.1 Creating the Import Package 19
7.1.2 Importing, Verifying, and Viewing the Package 20
7.2 Manufacturing End Devices 24
7.2.1 Running the Client to Provision a Node 24
7.2.2 Configuration ID and Configuration Alias 26
7.2.3 Node Root Key Creation 26
7.2.4 Batch Number 26
7.2.5 Troubleshooting 27
7.3 Exporting Key File for Devices 29

Appendix A Backup 31

A.1 Backing Up the Provisioning Client 31
A.2 Backing Up the Provisioning Server 31
A.3 Backing Up the End Device Output Data 31
A.3.1 End Device Configuration Information 31
A.3.2 End Device Key and Provisioning Data 32

Appendix B Node Message Encapsulation 33

B.1 Python Transport Wrapper 34
B.2 Code Examples 34
B.2.1 Python Transport Wrapper Pseudo-Code 35
B.2.2 Host Platform Pseudo-Code 36

Appendix C Host Provisioning 38

C.1 Extending the On-Ramp Wireless Provisioning Client 38
C.2 On-Ramp Wireless Provisioning Server 38
C.3 Performing Node Provisioning in Extended Tool 39
C.4 Retrieving Host App_Info Data 39
C.5 Saving Host Provisioning Outputs to the Server 40

Appendix D eMCM Provisioning 42

D.1 Installation 42
D.2 Operation 42
D.2.1 Single-stage Provisioning Using emcm_provision.exe 43
D.2.2 Multi-stage Provisioning Using emcm_factory.exe 44
D.3 Meter-specific Usage and Factory Procedures 47

Appendix E Abbreviations and Terms 48

Figures

Figure 1. Overview of the Provisioning Process 3
Figure 2. Network Key Generation and Distribution 5
Figure 3. End Device Key Generation and Distribution 6
Figure 4. Encapsulation Architecture 33
Figure 5. Encapsulation Example 34

Tables

Table 1. Software Compatibility Matrix 1
Table 2. Configuration Questions and Responses 13
Table 3. List of Typical Required Parameters 25
Table 4. Provisioning Client Return/Failure Codes 27
Table 5. Corrective Action Code Definitions 28
Table 6. List of Interface Arguments 39
Table 7. List of Interface Parameters 41
Table 8. Communication Options for Single-stage Provisioning 43
Table 9. Final Configuration Options for Single-stage Provisioning 43
Table 10. Diagnostic Output and Logging Options for Single-stage Provisioning 43
Table 11. Miscellaneous Options for Single-stage Provisioning 44
Table 12. Available Operations for Multi-stage Provisioning 44
Table 13. Communication Options for Multi-stage Provisioning 45
Table 14. Final Configuration Options for Multi-stage Provisioning 45
Table 15. Diagnostic Output and Logging Options for Multi-stage Provisioning 46
Table 16. Miscellaneous Options for Multi-stage Provisioning 46

Revision History

Revision	Release Date	Change Description
A	June 10, 2013	Initial release.
B	June 14, 2013	Added error code to troubleshooting section.
C	April 16, 2014	■ Added a new appendix containing provisioning information for eMCMs. ■ In the Troubleshooting section, added Return Code 72 to the table for Provisioning Client Return/Failure Codes.

1 Introduction

This document describes the setup, configuration, and provisioning of Nodes with security keys during production. This involves:

- Installing, configuring, and maintaining a provisioning server instance
- Optional development of application-specific software to support application host configuration and provisioning
- Installing provisioning clients (either directly or wrapped in a higher level application provisioning package) on manufacturing host PCs
- Preparing the provisioning server to support a target device configuration and On-Ramp network by importing configuration description files ("tarballs")
- Manufacturing and provisioning of end devices
- Exporting end device configuration and key data from the provisioning server and transmitting this data to the network operator

NOTE: This guide is intended for system administrator level users with root privileges. General familiarity with security concepts is required.

The following table indicates compatibility between software versions.

Table 1. Software Compatibility Matrix

	CommSys 1.X Node 1.X	CommSys 2.X Node 1.X	CommSys 2.X Node 2.X
Provisioning 1.5 (includes LKS, NPT, and KMS utilities)	✓	✓	
Provisioning 3.0	✓	✓	✓

2 Provisioning Overview

This section provides an overview of the provisioning process associated with the configuration or firmware upgrade of an end device such as an Advanced Metering Infrastructure (AMI) meter, Fault Circuit Indicator (FCI), etc. incorporating an On-Ramp Wireless Total Reach communications module. In this manual, we use the term “end device” to encompass both the On-Ramp Wireless communication node (e.g., a microNode or dNode) and any additional host applications (e.g., Meter Communications Module (MCM) in the case of an AMI meter) that need to be provisioned or upgraded. Provisioning and configuration of third party application firmware, such as an FCI host application, is the responsibility of the end device designers. However, some features of the On-Ramp Wireless provisioning system may be useful. See Appendix C: Host Provisioning for a discussion of available services.

The key components of the provisioning process are:

- Provisioning server
- Provisioning client
- Packaging server

2.1 Provisioning Server

The provisioning server is an integral part of key provisioning and management and is owned by the entity responsible for device provisioning. The provisioning server uses HTTP to communicate with one or more provisioning clients. The provisioning server maintains a database containing all necessary provisioning-related information associated with the end device:

- The Gateway root and code download (CDLD) keys
- Node configuration files
- Application-specific configuration files
- KMS and backend server public keys

This information is imported from configuration description files (tarballs) generated by the packaging server and is stored in a disk repository and database. The provisioning server is also used as a repository for storing the unique keys provisioned on each device by the provisioning client. These keys are stored in encrypted format and subsequently exported to a file that can be parsed by one or more associated backend network application servers such as the Key Management Server (KMS).

2.2 Provisioning Client

The provisioning client is typically a PC used to program, configure, and provision the end devices and is owned and managed by the entity responsible for device provisioning. Provisioning is typically done through a host board/host application with a serial port or Ethernet access. The provisioning process on the client is as follows:

1. The provisioning client requests the necessary configuration information from the provisioning server over HTTP.
2. Node firmware is updated, if necessary, and a target network-specific configuration is applied.
3. The provisioning client generates the unique device-specific security keys (e.g., node root keys, MCM security keys) and programs them to the device.
4. The provisioning client encrypts the unique, device-specific keys and relays this information to the provision server for storage.

The following information is typically programmed on the end device:

- Gateway key
- Code download (CDLD) key
- Node-specific root key
- Node configuration including AP list
- Host application security keys (optional)
- Host application configuration (optional)

Multiple provisioning clients can operate simultaneously to provision end devices in parallel.

2.3 Packaging Server

The packaging server is owned and managed by On-Ramp Wireless. It is used as a repository for updating and storing configuration information associated with an end device and the On-Ramp Total View Network. The configuration files are generated based on information received via an out-of-band method (e.g., email notification or XML-based web page form submission) which specifies the characteristics of the end device such as type (e.g., residential AMI meter with configured options or FCI) and deployment details such as operator info and/or deployment region.

The necessary information is packaged for distribution to provisioning server(s) as a single tarball and is identified by a unique 64-bit ID (provConfigId). The packaging server is responsible for version control and management of the tarballs. The following diagram provides an overview of key components and information flow involved in the provisioning process.

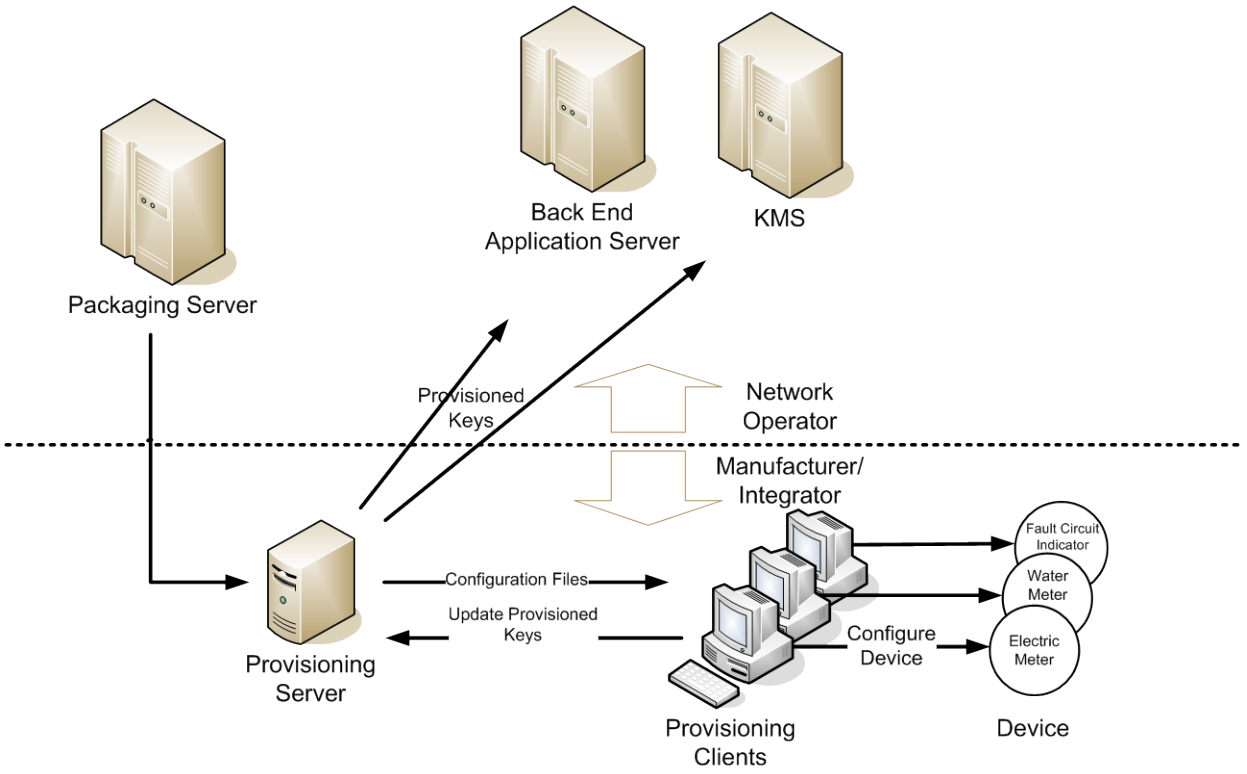


Figure 1. Overview of the Provisioning Process

3 Key Management Overview

3.1 Key Generation and Key Management

As part of the On-Ramp Wireless security solution, the node (and in some cases the host application) are programmed with a set of unique and shared security keys. These keys are never transmitted over-the-air and are used to support symmetric encryption and authentication procedures. In particular, the node is provisioned with network-specific keys and a unique, node-specific key. In addition, the host application may also be provisioned with a unique application key. As mentioned earlier, these node/device keys are generated by the provisioning client and, after provisioning is complete, need to be securely delivered to the peer backend application server (e.g., KMS for node keys, Head End System (HES) for MCM keys, etc.). The remainder of this chapter provides a brief overview of how the network keys are generated and managed.

3.2 Node Key Provisioning

This section provides a high level overview of node key provisioning.

1. A network deployment engineer uses the Generate Gateway Keys Utility (generate_gw_keys.py) to generate network-specific gateway keys while performing the initial installation of an On-Ramp Wireless network. This utility also creates an encrypted and signed output file that contains the Gateway keys for delivery to one or more provisioning servers. The resulting files are stored at the packaging server for inclusion in tarballs describing end devices for this network.
2. A tarball containing all configuration files necessary to provision an end device is generated, assigned a unique 64-bit identifier (provConfigId), and then distributed to the device provisioning server(s). This tarball contains the following files:
   - KMS public key
   - Backend server RSA public keys (if applicable)
   - Encrypted and signed Gateway and Code download(CDLD) key file
   - Node firmware binary file and associated hash
   - Node configuration files and associated hashes
   - Optionally, one or more application host firmware binary files and associated hashes
   - Optionally one or more application configuration files and associated hashes
   - An XML file describing the contents of the tarball in a form allowing parsing by a provisioning server
3. The provisioning server imports and verifies the provisioning package via a web interface. The server extracts the files and decrypts the Gateway keys using the user-entered passphrase. The passphrase for the provisioning server must be the same as the passphrase used to encrypt and sign the Gateway key file and is provided via an out-of-band procedure.

4. For each end device that will be provisioned, the provisioning client requests information required to provision the device. After successfully retrieving the required files and verifying the hashes, the provisioning client provisions the device such as:
   - ❒ Upgrades firmware as necessary
   - ❒ Updates device configuration
   - ❒ Generates device-specific securities keys that always include a node key and may include application keys
   - ❒ Writes the keys to the end device
5. After successfully completing the provisioning process, the provisioning client sends the device-specific keys to the provisioning server. Before transmission, the provisioning client encrypts the keys using the RSA public key associated with the peer backend application server.
6. The provisioning server stores the results of end device provisioning. At some point (generally after a set of devices has completed provisioning prior to shipping), an operator uses the provisioning server web interface to select a batch of devices and generate an XML file containing the encrypted device-specific keys and relevant configuration information. This file is conveyed to the network operator.
7. The KMS, or other backend key server, ingests the exported file containing the keys in a server-specific operation.

The following figures provide an overview of network key and device key generation and usage.

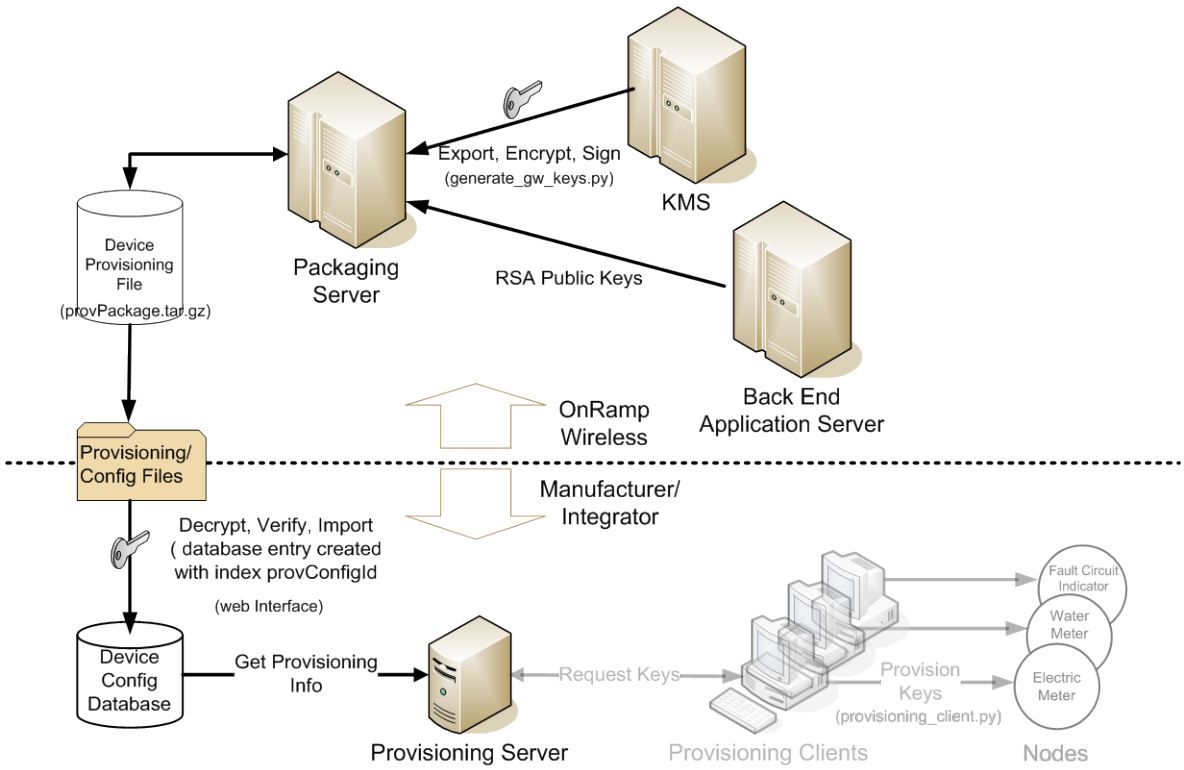


Figure 2. Network Key Generation and Distribution

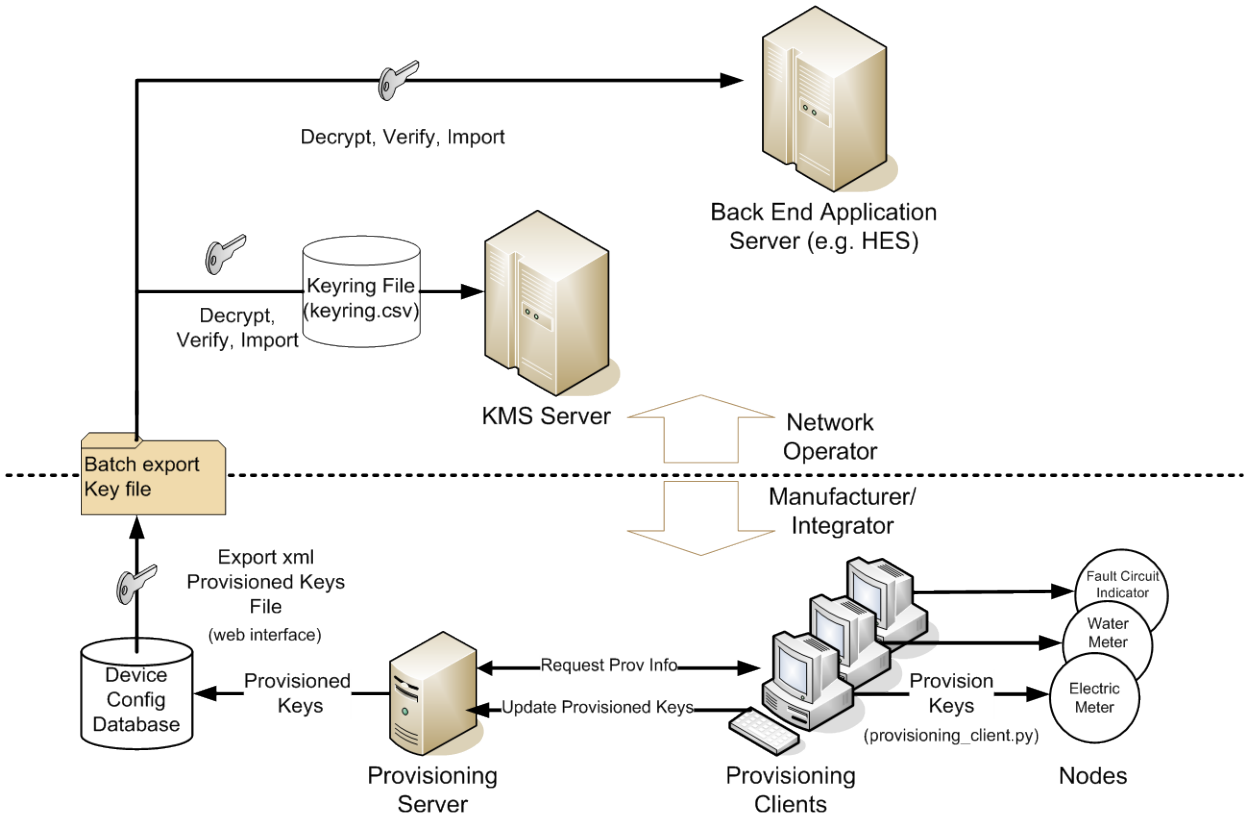


Figure 3. End Device Key Generation and Distribution

3.3 KMS Key Generation and Import Node Keys Process

The Gateway key and the code download (CDLD) key are unique to the network operator and are generated at network deployment time. These two keys are exported to the Gateway key file which is encrypted using the symmetric key shared between the KMS and provisioning server and signed using the KMS private key. The resulting gw_keys.csv and gw_keys_csv.sig files are exported to the packaging server for inclusion in all associated tarballs.

The KMS maintains the network keys and the node root keys for all authorized nodes on the system. As nodes are manufactured, they are provisioned with security keys. These security keys are exported from the provisioning server to an XML file (referred to as the batch export key file) at user-defined intervals.

When the KMS receives the batch export file from the provisioning server, it does the following:

- Extracts the relevant node data (i.e., it ignores all application-specific information)
- Decrypts the key information
- Adds the node keys to the active key database

For further details about the Key Management Server (KMS), refer to the *KMS User Guide (010-0062-00).*

4 System Requirements

4.1 Provisioning Server

System *requirements* for a provision server are:

- An enterprise-level server running CentOS 6.1 or later, or Red Hat® Enterprise Linux® (RHEL) 6.1 or later operating system
- Oracle/Sun Java 1.6 or 1.7
- MySQL or Oracle database instance
- Java Cryptography Extension (JCE);unlimited strength version
- Disk storage for provisioning data.

Refer to chapter 5 for details on installing Java, MySQL, and JCE as needed.

4.2 Provisioning Client

With some combinations of host operating systems and serial hardware, errors are occasionally observed in the serial port data. Reducing the serial baud rate tends to help reduce such errors but does not necessarily eliminate them. Some end devices may use a reliable delivery mechanism allowing for detection of errors and retransmission of data, generally by using one of the reliable transport wrappers (see Appendix B for details). The end device and provisioning client configurations must match in order to use such a protocol, and thus must be considered in the research and development phase of the end device design.

See chapter 6 for installation details.

4.2.1 Windows-based Platforms

System requirements for a provisioning client under Windows are:

- A PC running Windows® XP or Windows® 7 operating system
- A serial port (or other communications interface) matching the physical interface of the device to be provisioned

  NOTE: If a USB-to-UART adapter is used, an adapter based on an FTDI2232D device is recommended. Internal testing discovered that adapters based on Prolific PL2303 devices experienced problems on platforms running Windows 7 operating systems and therefore are not recommended.

4.2.2 Linux-based Platforms

System requirements for a provisioning client under Linux are:

- A PC running Ubuntu, Red Hat® Enterprise Linux® (RHEL) 6.1 or later, or CentOS/Scientific Linux 6.1 or later operating system. Others may work but are untested by On-Ramp Wireless.
- Python 2.6 or 2.7
- PySerial
- PyCrypto 2.5 or later
- A serial port (or other communications interface) matching the physical interface of the device to be provisioned

4.3 Shared File Repository

The provisioning server needs to serve files (e.g., binary, configuration, public keys, etc.) to the provisioning clients. The server populates these files (at tarball import time) to a directory tree in a location chosen by the installer which allows you to specify the top directory in which to place the provisioning configuration files for the *Provisioning Server*. Refer to Table 2: Configuration Questions and Responses.

Two operating modes exist to provide file access to the clients:

1. A standard IT shared network file system (SAMBA, NFS, etc.)
2. Direct HTTP file serving from the server to the client
   This is simple to set up but may incur a slight performance penalty if a large number of clients are in use.

The mode is chosen during server installation and is common to all clients of that server. See the "Use HTTP File Transfer" option in Table 2: Configuration Questions and Responses. Select "yes" to directly serve files and "no" to use a network file system.

If the system is using direct HTTP file transfers, no further configuration is necessary. Otherwise, the clients require read access to the storage location for provisioning files. It is often the case that the clients may need to use a different root directory path to access the files (e.g., a mapped drive letter or URI notation for server share access). This is supported by providing a client view of the directory path during installation of the provisioning server which allows you to specify the top directory in which to place the provisioning configuration files for the *Provisioning Client*. Refer to Table 2: Configuration Questions and Responses.

The most typical setup to satisfy these criteria is to store the files on local disk at the provisioning server, then configure SAMBA or NFS network file sharing protocols to allow access from the clients. SAMBA is more typical for Windows clients and NFS for Linux clients, although other configurations are possible. See the Red Hat administration documentation for help in configuring file sharing.

5 Installing and Configuring a Provisioning Server

This section describes the procedure for installing and configuring a provisioning server on a Linux platform.

NOTE: The person performing this installation ***must*** have root privileges on the server.

5.1 Prerequisite Packages for the Provisioning Server Installation

Prior to installing the Provisioning Server on the computer, the following software packages must first be installed.

5.1.1 Java

The Java version should be no earlier than 1.6. There are various ways to install Java. The following steps provide Java installation examples for two types of Linux platforms:

Scientific Linux (64-bit)

1. Go to the Oracle download site:

   http://www.oracle.com/technetwork/java/javase/downloads/jdk6u37-downloads-1859587.html

2. Accept the license agreement and then download the following file:

   ```
   jdk-6u37-linux-x64-rpm.bin
   ```

   NOTE: If Oracle changes this URL at any time in the future, go to the Oracle download site to find the installation package.

3. Execute the commands below in the order shown. Multiple RPM files are produced.

   ```
   chmod 700 jdk-6u37-linux-x64-rpm.bin
   sudo ./jdk-6u37-linux-x64-rpm.bin
   ```

4. Execute the following command to install Java:

   ```
   sudo yum localinstall jdk-6u37-amd64.rpm
   ```

Ubuntu

Use the following command to install Java:

```
sudo apt-get install sun-java6-jdk sun-java6-jre
```

5.1.2 MySQL Server

Prior to installation of the Provisioning Server, MySQL server must be installed either on the computer containing the Provisioning Server or on a computer that the Provisioning Server can access. The following steps provideMySQL installation examples for two types of Linux platforms.

Scientific Linux

1. Execute the following command to determine whether MySQL server has been installed on the computer:

   ```
   sudo service mysqld status
   ```

2. If the result is “unrecognized service,” the MySQL server has not been installed. Execute the following command to install MySQL server:

   ```
   sudo yum –y install mysql-server
   ```

Ubuntu

1. Execute the following command to determine whether MySQL server has been installed on the computer:

   ```
   sudo service mysqld status
   ```

2. If the result is “unrecognized service,” the MySQL server has not been installed. Execute the following command to install MySQL server:

   ```
   sudo apt-get install mysql-server
   ```

5.1.3 Java Cryptography Extension

Update Java Cryptography Extension (JCE) with Unlimited Strength Jurisdiction Policy Files. To do this, follow the instructions below.

1. Go to the Oracle Java download site:

   http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html

2. Scroll down to the bottom of the web page to find the row indicating “Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 6.”
3. Click on the ***Download*** button on the right side.
4. On the download page, accept the license agreement and then download the following zip file:

   ```
   jce_policy-6.zip
   ```

5. Unzip the file and go to the directory ’policy-6/jce.’

6. The jar files in this zip package need to be installed in the appropriate subdirectory of the installed Java version:
   `/<installation-dependent location>/jre/lib/security`

   For Red Hat/CentOS/Scientific Linux 6.x servers, this will be a location similar to the following:
   `/usr/java/jre1.6.0_27/lib/security`

   To discover the correct path, you can run:
   `java –version`

   Using the example above, this returns the following:
   `java version 1.6.0_27`

   If the correct location cannot be found, you can discover the Java binary location by running "`which java`" and then following the soft links until the actual binary is found. For example:

```
# which java
/usr/bin/java
# ls -las /usr/bin/java
       0 lrwxrwxrwx. 1 root root 22 Apr 11 09:53 /usr/bin/java ->
/etc/alternatives/java
# ls -las /etc/alternatives/java
       0 lrwxrwxrwx. 1 root root 30 Apr 11 09:53 /etc/alternatives/java ->
/usr/java/jre1.6.0_27/bin/java
# ls -las /usr/java/jre1.6.0_27/bin/java
       52 -rwxr-xr-x. 1 root root 50794 Jul 19  2011
/usr/java/jre1.6.0_27/bin/java
# ls -las /usr/java/jre1.6.0_27/
bin/                          lib/                          plugin/
Welcome.html
COPYRIGHT                     LICENSE                       README
javaws/                       man/
THIRDPARTYLICENSEREADME.txt
# ls -las /usr/java/jre1.6.0_27/lib/security/
total 136
 4 drwxr-xr-x.  2 root root  4096 Apr 11 09:53 .
 4 drwxr-xr-x. 18 root root  4096 Apr 11 09:53 ..
 4 -rw-r--r--.  1 root root    92 Jul 19  2011 blacklist
80 -rw-r--r--.  1 root root 81202 Jul 19  2011 cacerts
 4 -rw-r--r--.  1 root root  2253 Jul 19  2011 java.policy
12 -rw-r--r--.  1 root root 11087 Jul 19  2011 java.security
 4 -rw-r--r--.  1 root root   109 Jul 19  2011 javaws.policy
 4 -rw-r--r--.  1 root root  2481 Apr 12 16:15 local_policy.jar
16 -rw-r--r--.  1 root root 14189 Jul 19  2011 trusted.libraries
 4 -rw-r--r--.  1 root root  2465 Apr 12 16:15 US_export_policy.jar
```

5.2 Installing the Provisioning Server

To install and configure the provisioning server:

1. Obtain and expand the provisioning server installation tarball using the following command:

   `tar –xf provisioning_unix_{version}_{build}.tar.gz`

2. Go to the installation directory as follows:

   ```
   cd provisioning_unix_{version}_{build}
   ```

3. If this is an initial installation, execute the 'setup.sh' file first as shown below. Executing this file checks to see if Java and MySQL server are installed. If this is ***not*** an initial installation, skip to step 4.

   ```
   sudo ./setup.sh
   ```

4. The shell script 'provision_unix_{version}.sh' installs both the user interface and the console. Execute the script as shown below. Note that adding the `-c` option installs the console.

   ```
   sudo ./provisioning_unix_{version}.sh -c
   ```

5. You are prompted to confirm whether to install the Provisioning Server on your computer. Type 'o' or press the Enter key to continue with the installation.

   ```
   $ sudo ./provisioning_unix_3_0_1.sh -c
   Starting Installer ...
   This will install On-Ramp Wireless Provisioning Server on your
   computer.
   OK [o, Enter], Cancel [c]
   ```

6. The installation should be running in console mode. For a first-time installation, you are prompted with the following question. If this is ***not*** a first-time installation, skip to step 7.

   ```
   This will install Provisioning Web Service version 3.0.1
   Where should On-Ramp Wireless Provisioning Server be installed?
   [/opt/provisioning/server]
   ```

   Press the 'Enter' key to install the Provisioning Server in the default directory: /opt/provisioning/server.

7. If this is an upgrade to the Provisioning Server, you are prompted with the following question:

   ```
   A previous installation has been detected. Do you wish to update
   that installation?
   Yes, update the existing installation [1, Enter]
   No, install into a different directory [2]
   ```

8. If you respond with 'No' by entering '2' or pressing the 'Enter' key, you are again prompted with the following question:

   ```
   This will install Provisioning Web Service version 3.0.1
   Where should On-Ramp Wireless Provisioning Server be installed?
   [/opt/provisioning/server]
   ```

   Enter the location where you'd like the Provisioning Server to be installed.

9. If you responded with 'Yes' by typing '1' or pressing the 'Enter' key, you are prompted with the following configuration questions:

Table 2. Configuration Questions and Responses

Configuration Question	Response

`Choose the type of the database for Provisioning data`
`Choose Database Type:`
`MySql 5.x or higher [1, Enter]`
`Oracle 11g or higher [2]`
	■ For a MySQL database, type '1' or press the 'Enter' key.
	■ To use an Oracle database, the computer must first be configured by the database administrator for the Provisioning Server.

`Configure properties for the connection to the database.`
`Host`
`[localhost]`
	■ If the database is running on the same computer as the Provisioning Server, press the 'Enter' key.
	■ Otherwise, enter the IP address of the computer where the database is running.

`Port` `[3306]`	■ Press the 'Enter' key to use the default port number of 3306. ■ If the database uses a different port, enter the appropriate port number.
`User` `[]`	Enter the database user name (e.g., root).
`Password`	Enter the database password for the user specified above.
`Confirm Password`	Re-enter the password entered above.
`Schema` `[]`	Enter the database schema name used for the Provisioning Server, for example, orw_prov.

`Configure the port you want your tomcat instance to use.`
`Port`
`[8085]`
	■ Enter the port number for the Provisioning Web Server.
	■ If the default port 8085 is not already used, press the enter key to use 8085.

`Use HTTP File Transfer:`	Choose whether to serve files to clients directly using HTTP or using a network file system. For more details, see section 4.3 Shared File Repository.

`Top directory for provisioning files.`
`Provisioning Server Top Directory`
`[/workspace/provisioning]`
	Specify the top directory in which to place the provisioning files as seen by the Provisioning Server. For more details, see section 4.3 Shared File Repository.

`Provisioning Client Top Directory` `[/workspace/provisioning]`	Specify the path that the provisioning clients can use to access the provisioning files. For more details, see section 4.3 Shared File Repository. **NOTE:** This is only applicable if a shared file system approach is selected instead of direct http service.

10. After responding to all configuration questions, you will receive the following messages:

```
Setup has finished installing On-Ramp Wireless Provisioning Server on your
computer.

If the database has not been created, please create the database with the
sql script /opt/provisioning/server/sql/mysql-prov-install.sql.
```

```
Starting the provisioning server on Linux, please use command
sudo /sbin/service orw-provisioning start
Finishing installation ...
```

The installation has finished. Note that the database must be initialized and the server started before using. For details, see section 5.3 Configuring the Database.

5.3 Configuring the Database

You must create the database for the provisioning server. This section only describes how to create a MySQL database. If you are using an Oracle database, please see your database administrator, as the computer must first be configured by the database administrator for the Provisioning Server.

1. Go to the SQL script directory as specified at the end of the installation.

   ```
   $ cd /opt/provisioning/server/sql
   ```

2. Assuming the MySQL user name is “root” and the password is “onramp,” create the MySQL database using the following commands:

   ```
   $ mysql –uroot –ponramp
   mysql>  create database orw_prov;
   mysql>  quit;
   $ cat mysql-prov-install.sql | mysql  –uroot  –ponramp orw_prov
   ```

   The database has now been created and configured.

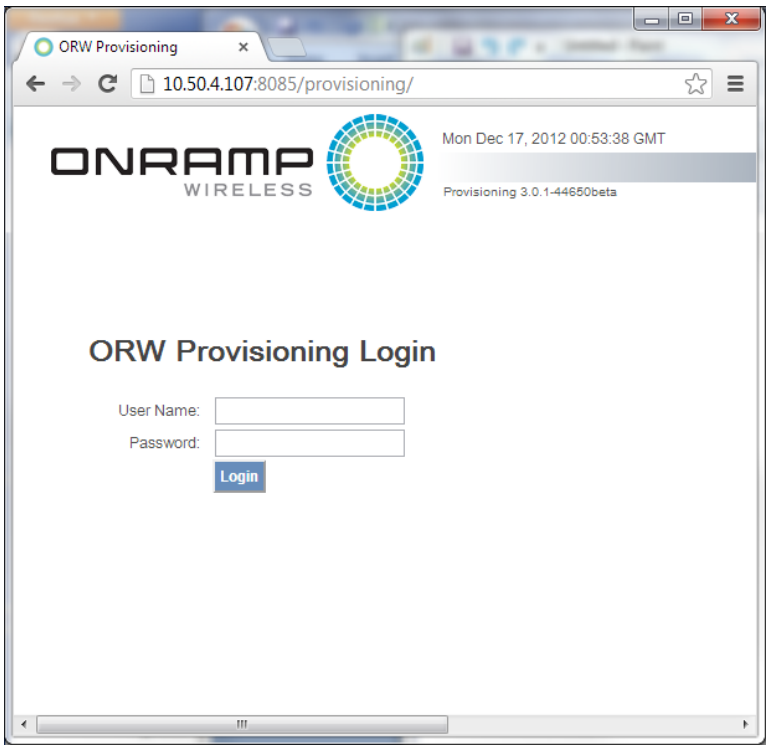
3. Start the Provisioning Server by entering the following command:

   ```
   $sudo /sbin/service orw-provsioning start
   ```

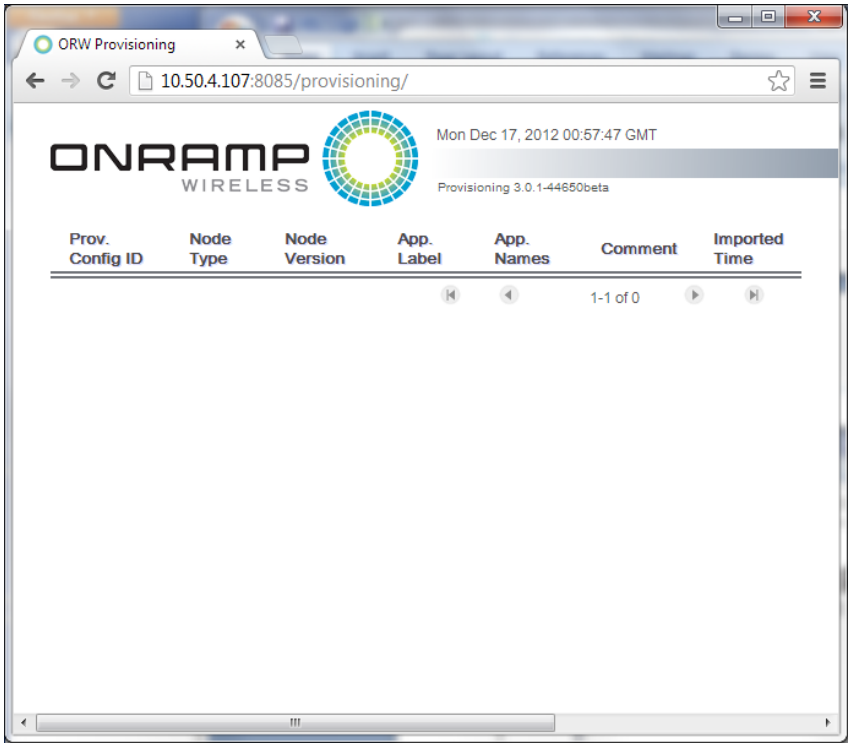
4. The Provisioning Server should be running. To verify that the server is running correctly, use a web browser to access the server. In the following example, for illustration purposes, we are using IP address 10.50.4.107 for the computer running the Provisioning Server. Type the URL in the web browser:

   http://10.50.4.107:8085/provisioning

5. If the server is running correctly, the login page should display as shown below.



6. Type the default user name ***admin*** and password ***onramp.*** Then click on the ***Login*** button***.*** The following screen should display as shown below.



7. The Provisioning Server has now been installed. For additional information and support for this functionality, contact On-Ramp Wireless at support@onrampwireless.com.

6 Installing and Configuring a Provisioning Client

The provisioning client does the following:

- Performs firmware updates, configuration, and security key provisioning for a node
- Communicates information relevant to the backend network back to a provisioning server for storage and later export

Running the client requires out-of-band communication of a unique 64-bit ID specifying the target network and configuration to be provisioned (or a string alias for this configuration ID), which must match an existing provisioning server configuration (created by ingesting a tarball with this configuration ID).

The client can run under Linux as a set of Python scripts. For supported Windows environments, the client is packaged as a monolithic Windows executable file with no supporting packages required.

6.1 Installation on a Windows-based Client

The Windows version of the provisioning client is distributed as a standalone zip file (e.g., provisioning_npt_windows_3.0.1.zip) and can be unzipped in a convenient location of the operator's choice. Unzipping the file produces a directory that includes a Windows executable file (provisioning_client.exe) that provides the provisioning client functionality. This executable can be run in a number of ways:

- Batch file
- Command line
- Shortcut
- Wrapper script
- Factory automation tool

6.2 Installation on a Linux-based Client

Ensure that Python version 2.6 or 2.7 is installed. You can check the Python version by typing the following on the command line:

```
python –-version
```

If Python is not installed or the incorrect version of Python is installed, see the package manager for your system to install either Python 2.6 or 2.7. In some cases, it may be necessary to install a parallel Python interpreter to avoid breaking other system tools. In this case, provisioning client invocation may need to explicitly run the appropriate version of the interpreter (version 2.6 or 2.7).

6.2.1 Installing PyCrypto on a CentOS 6.x/RHEL 6.x System

To install PyCrypto on a CentOS 6.x/RHEL 6.x system for the Python 2.6 installation, follow the steps below.

1. Install the Python development headers as follows:
   ```
   sudo yum install python-devel
   ```
2. Download PyCrypto 2.6 as follows:
   ```
   wget http://www.pycrypto.org/files/pycrypto-2.6.tar.gz
   ```
3. Extract PyCrypto using the following command:
   ```
   tar -zxvf pycrypto-2.6.tar.gz
   ```
4. Change directories to the extracted directory as shown below.
   ```
   cd pycrypto-2.6
   ```
5. Install PyCrypto directly by typing the following:
   ```
   python setup.py install
   ```
   The result should display the following information as the last two lines of error-free output.
   ```
   running install_egg_info

   Writing /usr/lib/python/site-packages/pycrypto-2.1.0-py.egg-info
   ```

6.2.2 Installing PySerial on a CentOS 6x/RHEL 6.x System

To install PySerial on a CentOS 6.x/RHEL 6.x system for the Python 2.6 installation, follow the steps below.

1. Download PySerial 2.6 as follows:
   ```
   wget http://pypi.python.org/packages/source/p/pyserial/pyserial-2.6.tar.gz
   ```
2. Extract PySerial using the following command:
   ```
   tar -zxvf pyserial-2.6.tar.gz
   ```
3. Change directories to the extracted directory as shown below.
   ```
   cd pyserial-2.6
   ```
4. Install PySerial directly by typing the following:
   ```
   python setup.py install
   ```
   The result should display the following information as the last two lines of error-free output.
   ```
   running install_egg_info

   Writing /usr/lib/python/site-packages/pyserial-2.6-py.egg-info
   ```

6.2.3 Installing PyCrypto and PySerial on an Ubuntu/Debian System

To install PyCrypto and PySerial on a system using the Advanced Packaging Tool (APT) (such as Ubuntu or Debian), type the following on the command line:

```
sudo apt-get install python-crypto python-serial
```

It may be necessary to upgrade PyCrypto by installing from source as indicated in section 6.2.2 above. Note that version 2.5 or later is required.

6.2.4 Installing the Provisioning Client

For Linux targets, the Provisioning Client is distributed in one of two ways:

- Tarball of Python files (provisioning_npt_3.0.1.tar.gz)

  The tarball form can be unpacked in a location chosen by the operator by using a command similar to the following:

  ```
  tar xvzf provisioning_npt_3.0.1.tar.gz
  ```

  This creates a directory called “npt” which includes a “provisioning_client.py” top level script. This top level script provides the Provisioning Client functionality.

- Architecture independent RPM (provisioning-npt-3.0.1.noarch.rpm) for RPM-based systems

  The RPM can be installed by using a command similar to the following:

  ```
  sudo yum localinstall provisioning-npt-3.0.1.noarch.rpm
  ```

  This command installs the Python scripts in /opt/provisioning/npt.

NOTE: Both installation methods (tarball and RPM) provide equivalent functionality.

7 Operation

Operation of the provisioning system consists of several different activities. As new networks are installed, new devices are introduced, or changes to device firmware or configuration occur, these changes must be tracked and reflected in updates or additions of new provisioning configurations.

These configurations are represented by “tarballs” of firmware, configuration files, Gateway/KMS key information, etc. These tarballs are managed internally by On-Ramp Wireless, and are supplied to end device manufacturing facilities via facility-specific file transfer mechanisms (web form, SFTP, email, etc.). These files are uniquely identified by a 64-bit provisioning configuration ID, allocated by On-Ramp Wireless. An operator at the manufacturing facility receives these files and imports them into the provisioning server through its web-based interface.

Once a provisioning configuration is created and imported to the server, provisioning clients can interact with an end device and the provisioning server to generate device-specific keys, perform necessary firmware and configuration updates, and supply required output information and keys to the server for later export.

After a batch of end devices is provisioned and ready for shipment to an end customer, the provisioning information and device keys must be exported from the server, via its web-based interface, then provided to the operator of the target network for import to the network key server(s).

7.1 Adding or Modifying a Device or Network

7.1.1 Creating the Import Package

On-Ramp Wireless is responsible for generating a tarball with the necessary files required by the provisioning server for successful provisioning of the end device. The first step in the process is to identify the characteristics of the end device as well as details about the operator network where the device will be deployed. This information is made available via a web page form, email, or other methods. After the information is acquired, the following items are collected for inclusion in the tarball.

- **KMS Public Key**
  This key is made available as part of ‘kms_key.pub.pem’ file. This information is obtained from the network operator.
- **Gateway-specific Keys**
  The encrypted Gateway-wide and code download keys are available in the ‘gw_keys.csv.aes’ file. In addition, the KMS authentication signature is available in the ‘gw_keys.csv.aes.sig’ file. The gateway-specific keys depend on the network/broadcast ID domains assigned to the operator’s network where the provisioned devices will be deployed.

- **Node Firmware Binary**
  The required node firmware depends on the following:
  - Whether the target system is 1.x or 2.x
  - The type of node (i.e., microNode/eNode/dNode)
  - Possibly on regulatory or customer/operator certification and acceptance testing

  For improved reliability, an MD5 hash across the node firmware binary file is also included in the tarball.
- **Node Field Configuration**
  The node field configuration file specifies device settings, the AP list, and system-wide parameter settings (e.g., system ID). On-Ramp Wireless and the operator work together to identify the network frequency plan. For improved reliability, an MD5 hash across the node field configuration file is also included in the tarball.
- **Node Factory Configuration**
  In the case where the integrator performs over-the-air verification at the factory site prior to shipping to the customer, the node factory configuration file specifies device settings, the AP list, and system-wide parameters (e.g., system ID) which are used for OTA testing done at the factory. On-Ramp Wireless and the integrator work together to specify the test network and network frequency plan. For improved reliability, an MD5 hash across the node factory configuration file is also included in the tarball.

In addition to node-specific files, depending on the end device, the tarball may include one or more sets of files related to application host provisioning. Analogous to the node-specific files, the following files are included in the tarball in order for each host application to be provisioned.

- Peer Backend Server Public Key
- Application Host Firmware Binary
- Host Field Configuration
- Host Factory Configuration

7.1.2 Importing, Verifying, and Viewing the Package

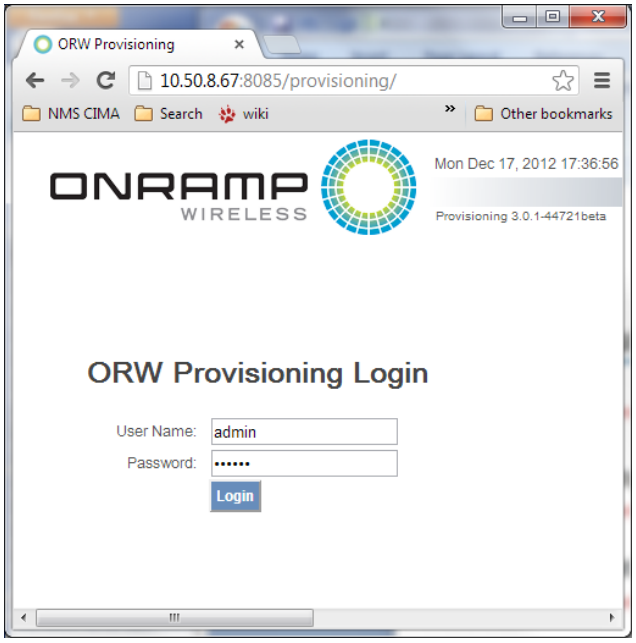
The configuration package must be imported into the Provisioning Server so that the Provisioning Client can access the configuration and files contained in the package. The following steps explain the process.

1. From the web browser, enter the URL for the Provisioning Server. For the purposes of this example, we'll use the following URL:

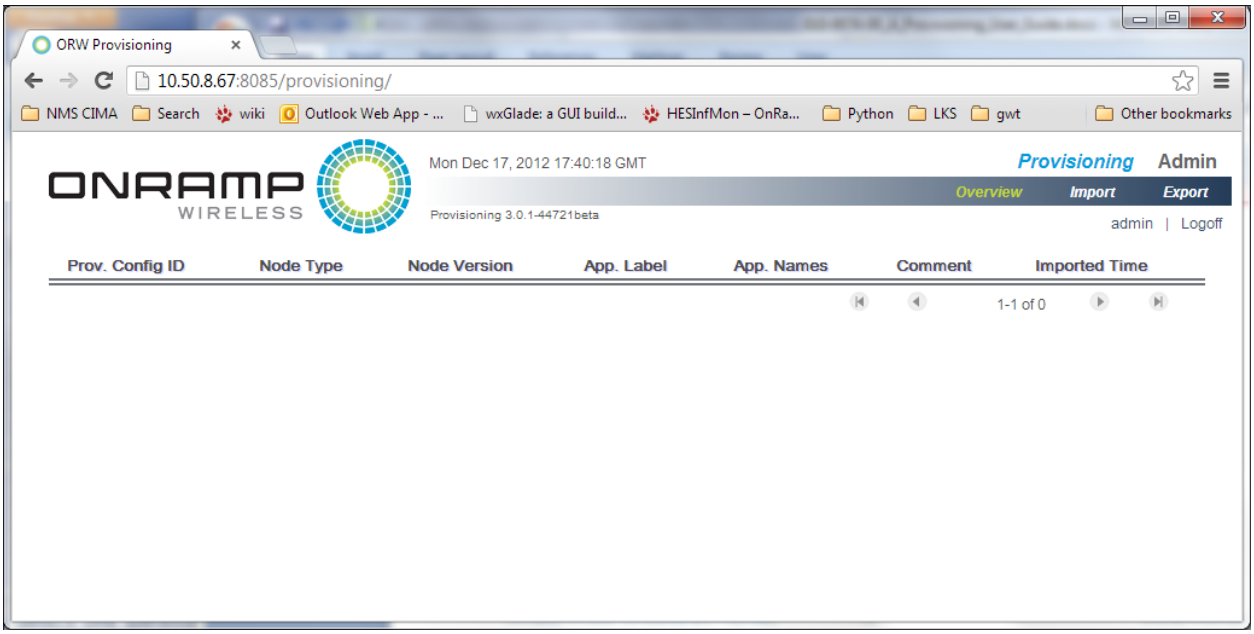
   http://10.50.8.67:8085/provisioning

2. Enter the user name and password to log into the Provisioning Server. The defaults are:
   - User name = admin
   - Password = onramp

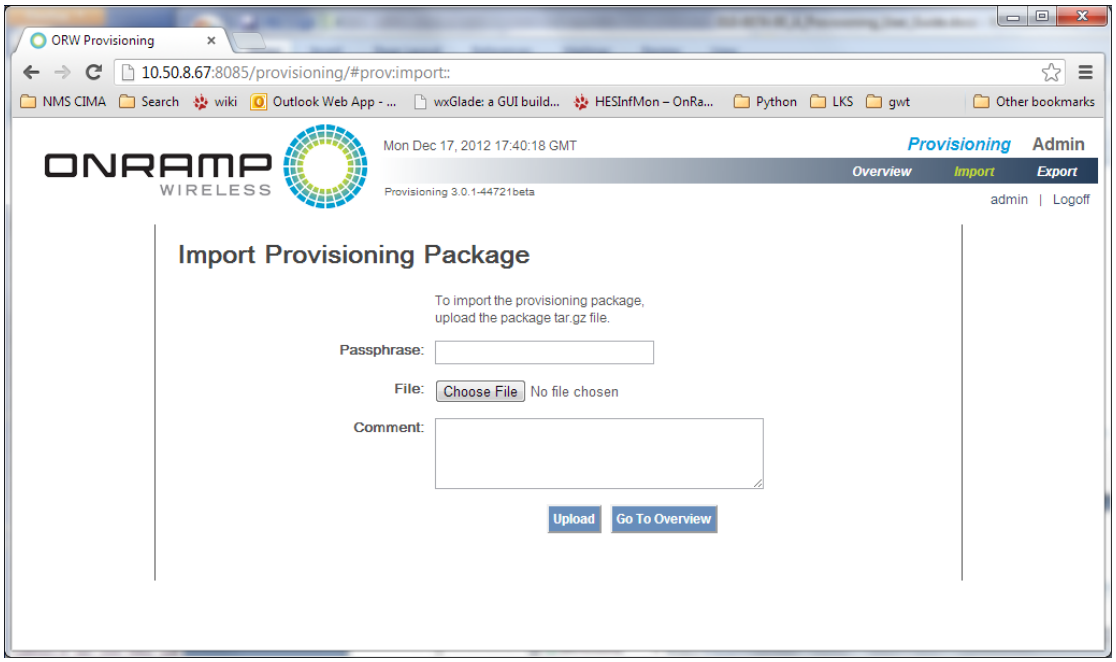
   The overview page (shown below) displays.



3. From the overview page, click on ***Import***, in the upper right corner.



4. The “Import Provisioning Package” page displays.



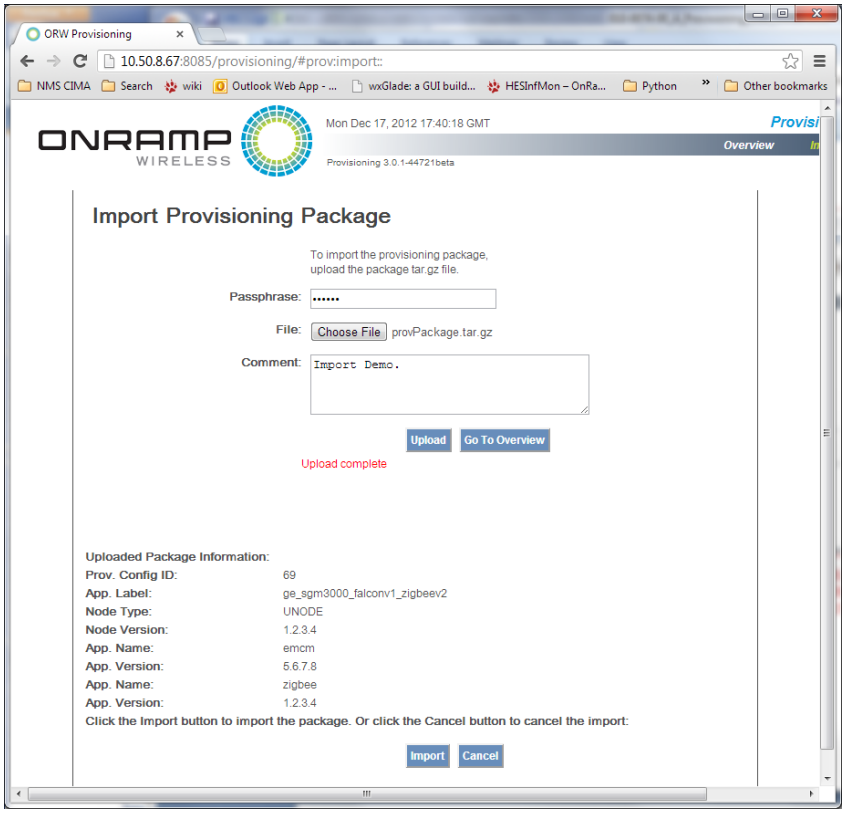
5. On the “Import Provisioning Package” page, perform the following steps:

   a. Enter the passphrase used to encrypt the gateway key file (selected at the time of network deployment and supplied with the tarball)

   b. Choose the import package file (tarball)

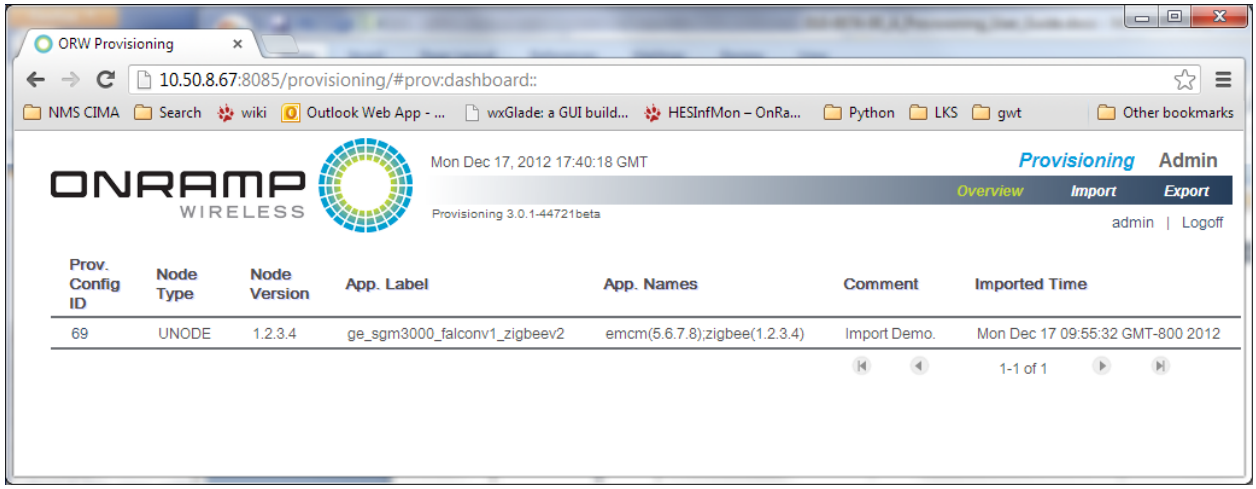
   c. Click on the ***Upload*** button.

   If the passphrase is correct and the import package is properly built, the “Import Provisioning Package” page shows the package information for the verification as shown below.

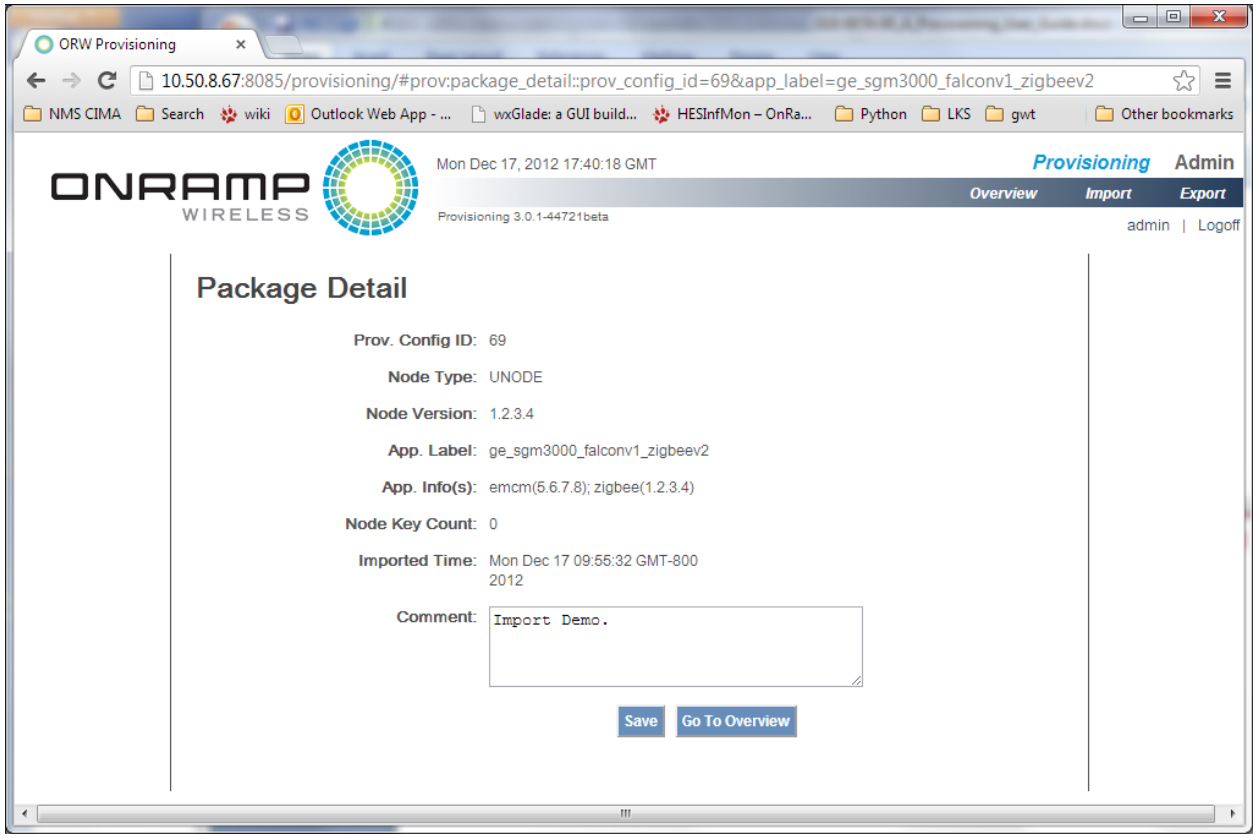


6. Next, click on the ***Import*** button to import the package.

7. After the package is imported, go to the overview page by clicking on ***Overview*** (in the upper right corner) to view the imported package list.



8. Click on the Prov. Config ID of the package to get more detailed package information as shown below.



7.2 Manufacturing End Devices

Provisioning an end device necessarily includes provisioning of the node resident in the end device, but may also require operations on the host processor. If applicable, these host operations are specific to the end device and require software development using the provisioning client as a dependency. See Appendix C for a discussion of options for host provisioning. The remainder of this section assumes host provisioning (i.e., firmware programming and any configuration of the host) are handled separately without depending on tools supplied by On-Ramp Wireless.

7.2.1 Running the Client to Provision a Node

Windows Environment

The provisioning client is an executable (provisioning_client.exe) produced from Python using the py2exe tool. It can be executed from a command prompt and launched by using a desktop shortcut or running a sub-process from a higher layer test framework.

The following example provides a typical command line under Windows:

```
provisioning_client.exe –d COM1 -s 192.168.0.10 -B 1000 –-config-id=0x12345678abc
```

This command allows the node to communicate with the provisioning server at IP address 192.168.0.10 and with the end device attached to serial port COM1. It also does the following:

- Upgrades or downgrades the node firmware to the required version, if necessary
- Writes the node configuration file specified by the server to the node
- Generates a unique node root key
- Writes the node root key and the configured gateway keys to the node
- Communicates the node ID and the new root key, which is encrypted with the target network KMS public key, to the server for storage until a batch export is done

Linux Environment

The Provisioning Client is a Python script that can be made executable and run directly from a command line. This assumes that the default Python interpreter is version 2.6 or 2.7 and is in the user's path. If this is not the case (i.e., if the default interpreter is not version 2.6 or 2.7) the client script can be passed as an argument to an appropriate Python interpreter. For example:

```
python2.6 ./provisioning_client.py <arguments>
```

The Provisioning Client is also designed to be embedded in other Python tools as a module that can be imported. For more information, see Appendix C: Host Provisioning.

Under either Windows or Linux, a number of arguments must be passed to the Provisioning Client to provide the information necessary for provisioning a node. There are also a number of optional arguments affecting its operation, allowing verbose operation, logging to a file, etc. For a complete list of options, refer to the Python Help by running the client with a –h or `--help` argument on the command line.

Table 3. List of Typical Required Parameters

Parameter	Description
`-d <serial_port>`	The port used for communicating with the node.
`--device=<serial_port>`	
`-s <ip_addr>[:<port>]`	The TCP port number of the Provisioning Server. **NOTE:** It is not necessary to specify the port if the default port of 8085 is in use.
`--server=<ip_addr>[:<port>]`	
`-B <provisioning_batch>`	The batch number for the current provisioning run. The batch number is an arbitrary integer number that can be assigned to each provisioning run of one or more nodes for tracking purposes. The batch number is saved in the server key database and can be used to select keys for key export.
`--batch=<provisioning_batch>`	
`-p <prov_config_id>`	The unique ID for the provisioning configuration—a 64-bit integer defining the target network, application, and configuration.
`--config-id=<prov_config_id>`	
`--config-alias=<config_name>`	A string name for the configuration that must be included in the configuration tarball so the Provisioning Server can map to a configuration ID.

7.2.2 Configuration ID and Configuration Alias

In some cases, a device manufacturer may have a system already in place to track end device configurations as they are manufactured. The provisioning system uses the 64-bit provisioning configuration ID (`-p` or `--config-id` in the client arguments) to identify a target device configuration.

If the manufacturer has an alternate configuration naming scheme, the mapping from the manufacturer name to configuration ID can be done in two ways:

- By the manufacturer at the time of device provisioning

  or

- If the manufacturer names are known in advance, they can be provided to On-Ramp Wireless at the time the tarball is created. In this case, the Provisioning Server can map a string (called a configuration alias) to a configuration ID.

In some cases, multiple manufacturer strings may correspond to the same provisioning configuration, so multiple configuration aliases can map to the same configuration ID.

NOTE: Since an alias will be mapped to a configuration ID by the server, only one of the following arguments should be specified when running the Provisioning Client:

```
–config-id
  or
–config-alias
```

7.2.3 Node Root Key Creation

The provisioning client creates node-specific root keys dynamically at the time of node key provisioning. These keys are encrypted with the KMS public key (retrieved from the provisioning server) before being transported back over the wire to the server for storage. At no point is this key saved to disk or sent over the socket to the server in plain text form. The Provisioning Server saves the newly generated and encrypted node-specific root key and the associated batch number into the Provisioning Server key database. Provisioning a node again simply generates a new random key, overwriting the old key.

7.2.4 Batch Number

The batch number is a user-determined, 32-bit, unsigned integer number that can be used to facilitate key export of a group of keys. Multiple nodes can be associated with the same batch number. The purpose of the batch number is to facilitate tracking and logical grouping of node keys. For example, the batch number can be a sales order number so that all nodes produced and provisioned for that sales order can be grouped and tracked together. If the sales order number is not a true integer but contains alpha-numeric characters, then it must be mapped to and associated with an integer batch number.

NOTE: Note that the batch number argument is initially treated as a string and converted to an integer value. A string argument that begins with a ‘0’ (for example, 05082011) is interpreted as an octal value and a string argument starting with ‘0x’ (for example, 0x1238ef) is interpreted as a hexadecimal value.

7.2.5 Troubleshooting

The following tables provide return/failure codes for the provisioning client, how to interpret the codes, corrective actions, and corrective action definitions.

Table 4. Provisioning Client Return/Failure Codes

Return Code	Interpretation	Corrective Action (See Table 5 for code definitions.)
0	Successful operation	N/A
2	Internal SW error (invalid logger instance)	11
3	Invalid command line arguments - check arguments against usage	2
4	Unable to communicate with device under test	2
14	Unknown error	10, 11
36	Message to node failed	11
37	Node returned unexpected error	11
38	Verification of node provisioning failed	3
39	Node version incompatible with provisioning version	11
40	Node HW type incompatible with provisioning version	11
41	Node calibration information incompatible with provisioning version	11
42	Node configuration information incompatible with provisioning version	11
43	Node firmware from prov. server unavailable or doesn't match node type	3
44	Node configuration from prov. server unavailable or invalid	3
68	Unable to communicate with device under test	2
69	Host (e.g., eMCM) returned unexpected error	11
70	Verification of host (e.g., eMCM) provisioning failed	3
72	Host (e.g., eMCM) reported failing or invalid hardware	11
75	Host (e.g., eMCM) firmware from prov. server unavailable or doesn't match host type	3, 5, 10
76	Host (e.g., eMCM) configuration from prov. server unavailable or invalid	3, 5, 10
100	Provisioning server unreachable	4
101	Provisioning server returned error response	3
102	Response from provisioning server invalid or file signature mismatches	3
108	Unable to open file from provisioning server	5, 10
132	Gateway unreachable	6
133	Gateway returned unexpected error	11
134	Device under test did not join gateway correctly	7, 8
140	Gateway message library not found	11
141	Gateway SSL certificates not found	11
164	Application-specific communication failure. For an eMCM, this code means that it cannot communicate with meter.	9

Table 5. Corrective Action Code Definitions

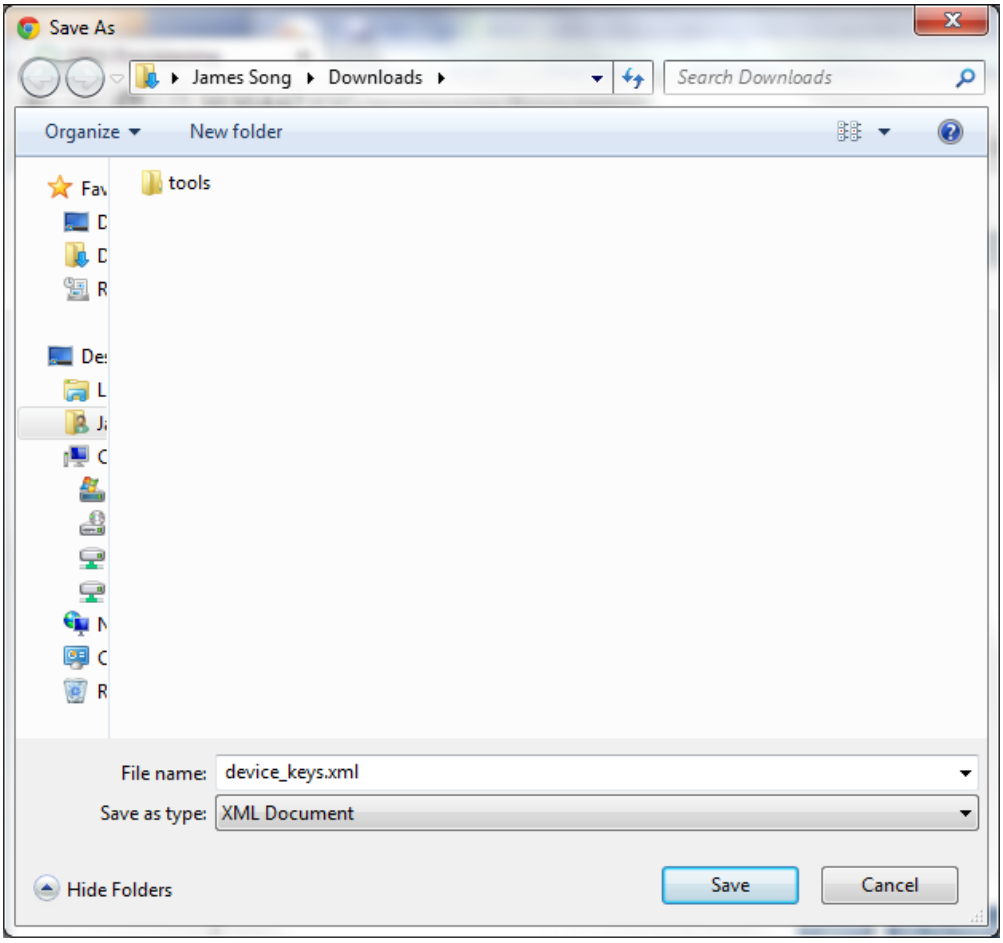
Corrective Action Code	Definition
1	Check/correct arguments supplied to provisioning client software.
2	Check that the communications cable is correctly installed, the COM port is correct, and that the DUT is powered up.
3	Check that the configuration alias or configuration ID is correct and that the up-to-date tarball is loaded at the provisioning server. Reload appropriate tarball if unsure.
4	Check that the provisioning server is running and accessible to the client machine (e.g., try to access server web page from the client machine).
5	If using shared file system mode, check that the client can access files on the shared drive.
6	Check that the gateway is running and that it can be reached from the client machine (e.g., ping the gateway server and check the EMS for gateway status).
7	Check that the AP is online and that the unit under test is in the vicinity of the AP. Check the AP web page for status.
8	Check that any external antenna is attached correctly.
9	Check that the eMCM module is seated in the meter correctly and that the meter is powered and configured to match the eMCM module.
10	If using the HTTP file serving mode, check that the provisioning client software has write permissions to its local directory.
11	Collect the provisioning client log file and send it to On-Ramp Wireless at support@onrampwireless.com.

7.3 Exporting Key File for Devices

1. To export the provisioned keys, go to the "Export Keys" page by clicking on ***Export*** in the upper right corner of the Provisioning Server web page.

2. Enter information into the relevant fields (i.e., Node ID, batch number(s), etc.) in order to select the devices to be included and then click the ***Export*** button at the bottom of the page.
3. The 'Download Dialog Box' displays for the file containing the keys to be exported (i.e., `file device_keys.xml`). This XML file contains the encrypted keys and relevant configuration information. Navigate to where you want to save this file and then click on the ***Save*** button.

   NOTE: The 'Download Dialog Box' varies depending on the browser and its settings. For the Chrome web browser (used in the example), the following screen displays:



4. The generated key file should then be conveyed to the target network operator to be ingested by the backend key server(s).

Appendix A Backup

Backing up provisioning data requires that you back up the following:

- The Provisioning Client
- The Provisioning Server
- The end device provisioning result data

A.1 Backing Up the Provisioning Client

For a Provisioning Client installation, no backup is necessary because no persistent data is stored. A failed machine can simply be replaced and the required software re-installed on the client.

A.2 Backing Up the Provisioning Server

The Provisioning Server saves the following information:

- Configuration information which is set up during installation
- Tarball file contents which are saved in the working directory specified during installation
- Tarball metadata in its database
- End device key and provisioning data in its database (MySQL or Oracle).

The server configuration and installation may be backed up by saving the /opt/provisioning/server directory on the server machine in whatever way makes sense for the local IT infrastructure. This is only for convenience and is not required. A re-install of the server from scratch will restore equivalent data.

A.3 Backing Up the End Device Output Data

A.3.1 End Device Configuration Information

The data content of ingested tarballs (end device configuration information) can be backed up by performing the following steps:

1. Duplicate the file storage directory (which is specified during installation of the provisioning server).
2. Back up the provisioning server database.

Alternatively, the input tarballs can be archived and later imported to a newly installed Provisioning Server instance. However, be aware that the order of import is important in the case of updates to the same configuration, so only the most recent versions of configuration description files should be imported.

A.3.2 End Device Key and Provisioning Data

To back up the end device key and provisioning output data, the database must first be backed up. See documentation for the database backend of choice (e.g., for MySQL, go to http://dev.mysql.com/doc/refman/5.1/en/backup-methods.html. This site documents a variety of backup approaches for MySQL).

Devices whose data has been exported to a key export file may not require backup. This assumes that the key files are either archived or the responsibility of the network operator to archive and back up. However, a database backup is a recommended failsafe in the case of mislaid data or for historical records.

Appendix B Node Message Encapsulation

On-Ramp Wireless provides a set of Python tools for managing, testing, and provisioning the Node. These tools support encapsulation of messages to the Node via transport wrappers in the case where the host platform has a transport layer built on top of the physical transport layer (usually Universal Asynchronous Receiver/Transmitter [UART]). By using encapsulation, the host platform can communicate to both its software and the Node without having to put the host platform into a special mode where it can only communicate to one or the other. Encapsulation also allows the host platform to use its own acknowledgement and retransmission mechanism in case the physical layer communication to the host platform is not 100% reliable.

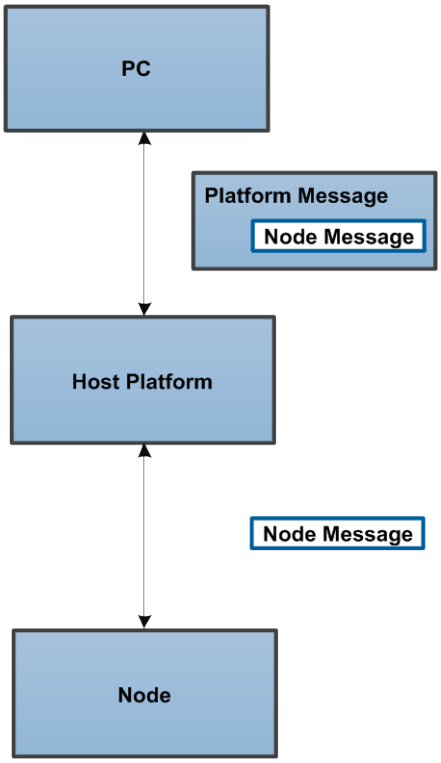


Figure 4. Encapsulation Architecture

The encapsulation can do any of the following (and more) to the message:

- Add a header or footer
- Disassemble or reassemble the message if there are packet size limitations communicating to the host platform
- Perform a cyclic redundancy check (CRC) to check the integrity of the message
- Acknowledge and retransmit lost or corrupt messages

To enable encapsulation, code must be written on both the PC side and the host platform.

B.1 Python Transport Wrapper

NOTE: Consult On-Ramp Wireless to determine whether your host requires message encapsulation. Hosts that require message encapsulation must use the following command line option: `--wrapper=<wrapper_name>.`

A limited set of existing wrappers are included in the Provisioning Client distributions. If an end device designer wishes to create a new wrapper, they must use a Python script client format or coordinate inclusion of a new wrapper in python_client windows executable distributions.

In the Provisioning Client Python distribution, the transport wrappers are located in a subirectory called 'transport_wrappers.' In this subdirectory, you will find:

- A transport wrapper stub in 'wrapper_orw.py'
- An implementation of a reliable transport over UART in 'wrapper_reliable.py' which is compatible with the RHT implementation included in legacy UNIL releases
- An updated and optimized version of the reliable wrapper in 'wrapper_rht_v2' which is compatible with the RHT implementation in host_cmn releases. **NOTE:** This should be used for new designs.

The wrapper is initialized with a read and a write function. The read function is called with a *decapsulated* message *from* the Node. The write function is called with an *encapsulated* message *to* the Node.

The wrapper must implement two functions:

- egressWrapper

  The Node Python tools call 'egressWrapper' in order to send a message to the Node. This function should encapsulate the message and then call the write function with which the wrapper was initialized.

- ingressWrapper

  The Node Python tools call 'ingressWrapper' when data is received from the host platform. When a complete message from the host platform is received, if the message is from the Node, the wrapper decapsulates it and calls the read function with which it was initialized.

To use the newly created wrapper (located at 'transport_wrappers/wrapper_foobar.py'), use the `–wrapper` option when calling the Python script at the command line.

For example:

```
./provisioning_client.py –wrapper=foobar …
```

B.2 Code Examples

Example code for implementing a simple form of encapsulation is diagrammed below.

Length (2 bytes)	0x47 (1 byte)	Node message (variable)	Checksum (1 byte)

Figure 5. Encapsulation Example

B.2.1 Python Transport Wrapper Pseudo-Code

```
receiveBuffer = “”

def checksum(str):
    checksum = 0
    for c in str:
        checksum = (checksum + ordI) & 0Xff
    return checksum

class TransportWrapper():
    def __init__(self, writeFxn, readFxn):
        self._writeFxn = writeFxn
        self._readFxn = readFxn
        pass

    def egressWrapper(self, str):
        # The length of the encapsulated message is 4 greater than the
size of
        # the Node message.  There are 2 bytes of length, one byte with
0x47 to
        # signify that this message is for the Node, and one byte of
checksum.
        MsgLen = len(str) + 4
        str = chr(msgLen & 0Xff) + chr(msgLen >> 8) + chr(0x47) + str
        str += chr(checksum(str))
        self._writeFxn(str)

    def ingressWrapper(self, str):
        global receiveBuffer
        receiveBuffer += str
        if len(receiveBuffer) >= 2:
            msgLen = ord(receiveBuffer[0]) + (256 *
ord(receiveBuffer[1]))
            if len(receiveBuffer) >= msgLen:
                # A full message has been received.
                Msg = receiveBuffer[0:msgLen – 1]
                rxChecksum = ord(receiveBuffer[msgLen – 1])
                calcChecksum = checksum(msg)
                if rxChecksum == calcChecksum:
                    if ord(msg[2]) == 0x47:
                        # Strip off header and pass up Node message.
                        Self._readFxn(msg[3:])
                    else:
                        # This message is not from the Node.  Ignore
it.
                        Pass
                    # Remove message from the receive buffer.
                    ReceiveBuffer = receiveBuffer[msgLen:]
                else:
                    # Checksum did not match.  Implement some kind of
recovery.
                    Pass
```

B.2.2 Host Platform Pseudo-Code

```
uint16_t foobar_rxBufLen = 0;
uint8_t foobar_rxBuf[512];
uint8_t foobar_txBuf[512];

uint8_t
FOOBAR_Checksum(uint8_t * msgPtr,
                uint16_t msgLen)
{
    int I;
    uint8_t checksum = 0;

    for (I = 0; I < msgLen; i++)
    {
        checksum += msgPtr[i];
    }

    return checksum;
}

void
FOOBAR_ProcessRxData(uint8_t * dataPtr,
                     uint16_t dataLen)
{
    memcpy(&foobar_rxBuf[foobar_rxBufLen],
           dataPtr,
           dataLen);

    foobar_rxBufLen += dataLen;

    if (foobar_rxBufLen > 2)
    {
        uint16_t msgLen = foobar_rxBuf[0] + (256 * foobar_rxBuf[1]);

        if (foobar_rxBufLen >= msgLen)
        {
            /* A full message has been received. */
            uint8_t rxChecksum = foobar_rxBuf[msgLen – 1];
            uint8_t calcChecksum = FOOBAR_Checksum(foobar_rxBuf,
                                                   msgLen – 1);

            if (rxChecksum == calcChecksum)
            {
                if (foobar_rxBuf[2] == 0x47)
                {
                    UNIL_PT_SendMsg(&foobar_rxBuf[3],
                                    msgLen – 4);
                }
                else
                {
                    /* This message is not for the Node. */
                    HOST_PLATFORM_ProcessMsg(foobar_rxBuf,
                                             msgLen);
                }
```

```
                /* Remove message from the receive buffer. */
                memmove(foobar_rxBuf,
                        &foobar_rxBuf[msgLen],
                        foobar_rxBufLen – msgLen);

                foobar_rxBufLen -= msgLen;
            }
        }
    }

    return;
}

void
FOOBAR_SendMsgToPc(bool nodeMsg,
                   uint8_t * msgPtr,
                   uint16_t msgLen)
{
    msgLen += 4;
    foobar_txBuf[0] = msgLen & 0Xff;
    foobar_txBuf[1] = msgLen >> 8;

    if (nodeMsg == true)
    {
        foobar_txBuf[2] = 0x47;
    }
    else
    {
        foobar_txBuf[2] = 0x00;
    }

    memcpy(&foobar_txBuf[3],
           msgPtr,
           msgLen – 4);

    foobar_txBuf[msgLen – 1] = FOOBAR_Checksum(foobar_txBuf,
                                               msgLen – 1);

    USB_TxBuf(foobar_txBuf,
              msgLen);

    return;
}
```

Appendix C Host Provisioning

In general, there is no requirement for an end device designer to use the On-Ramp Wireless provisioning system to perform host provisioning for:

- Updates of the host system including host configuration and firmware
- Generation and programming of a host security key

Design of this process is the responsibility of the end device designer(s). However, since the node requires provisioning time updates (particularly for target system ID, AP list, and security keys), On-Ramp Wireless has implemented infrastructure to support factory operations. This support may help the end device designer by allowing a bundled configuration, firmware, etc. for both the node and the host platform to be managed and tracked together. Furthermore, key generation and delivery of keys back to the end customer can be shared.

C.1 Extending the On-Ramp Wireless Provisioning Client

To take advantage of the provisioning infrastructure, the end device designer must extend the Provisioning Client software to perform host application provisioning. The On-Ramp Wireless Provisioning Client software is implemented in the Python programming language, so host provisioning software must interoperate with Python in order to leverage On-Ramp Wireless tools. The Provisioning Client is architected to cleanly embed into an overall end device provisioning tool implemented in Python or with a Python interoperability layer.

NOTE: In the following discussion, familiarity with the Python programming language is assumed. A starting point for language instruction can be found at http://docs.python.org/2/tutorial/.

The Python module provisioning_client.py can be imported, its interfaces called to perform node provisioning, and its code may be emulated to provide for host provisioning.

C.2 On-Ramp Wireless Provisioning Server

In general, the On-Ramp Wireless Provisioning Server maps a descriptive ID (64-bit integer called provConfigID) into a list of provisioning information bundles, each of which contains configuration, firmware, and a key information bundle which is specific to the node. However, an end device designer can specify N (most typically one, but more are supported) bundles of application information. Each app_info instance consists of the following:

- One firmware binary
- One configuration file to be used during manufacturing
- One configuration file to be loaded at the end of provisioning for customer operation
- An RSA public key to be used to encrypt any cryptographic key information related to the app instance

Any of this information may be ignored by the end device designer.

C.3 Performing Node Provisioning in Extended Tool

If the end device designer chooses to wrap the Provisioning Client in a higher layer tool, the overall tool still needs to provide for node provisioning. This can be done either by launching the provided Provisioning Client as a standalone process to handle node provisioning, or by importing the provisioning_client.py file, then calling the provisioning_client.provision() interface with appropriate arguments. In particular:

Table 6. List of Interface Arguments

Argument	Description
`conn`	An object encapsulating a communications path to the node. This must be an instance (or have the same interface as) a `ucl_nhp.Connection()` class. See the `cmdLineMain()` function of provisioning client.py for sample code to correctly initialize such an object.
`logger`	A Python logging.logger instance for diagnostic output. To set up a logger, see the setupLogger() function of provisioning_client.py for sample code.
`debug`	A Boolean indicating whether diagnostic output should be produced.
`config_id`	The 64-bit provisioning configuration identifier.
`server_ip_addr`	The IP address or host name of the Provisioning Server instance.
`server_port`	The TCP port of the Provisioning Server instance. The default is 8085, but the TCP port must match the server installation.

Other parameters can be used to fine-tune operation but these parameters have typical default values.

C.4 Retrieving Host App_Info Data

Wrapping the Provisioning Client in a higher level tool facilitates host updates or configuration in the factory. The host provisioning tool can retrieve firmware and configuration files from the Provisioning Server and save key or other application-specific data to the Provisioning Server for later export and ingest by the backend network. This is accomplished by communicating with the server using the messaging primitives in npt_comm.py. In particular, a server communications object is created by code similar to the following example:

```
prov_srv = npt_comm.ProvisioningServerInterface(
    logger, server=s_ip, port=8085)
```

where:

- `logger` is a Python logger instance
- `s_ip` is the IP address or hostname of the provisioning server
- `port` is the TCP port of the provisioning server

Given a provisioning server instance, the interface getConfig() may be used to retrieve the firmware and configuration file information. The first two arguments relate to node type and ID, and may be set to “None” for host information only access:

```
cfg = prov_srvr.getConfig(None, None, config_id)
```

where `config_id` (or `network_id` in some provisioning releases) is the 64-bit provisioning configuration ID.

The cfg object returned is an object of type npt_comm.ConfigData(). The host portions of the configuration are represented by the member lists:

- **app_names**

  In the typical case where the host requires one firmware binary, one config file, and optionally a second config file for factory use, the length of all these lists should be one. For a complex host with more than one processor or multiple binaries to be loaded, multiple instances of this data is provided, in which case the list lengths will be > 1. In this case, the app_names list entries are arbitrary strings allowing the provisioning tool to differentiate between the different provisioning targets.

- **app_versions**

  The app_versions list contains a string indicating the version of the firmware included – this can be used to optionally skip firmware updates if the host is already up to date, or can be used for a sanity check that the upgrade process succeeded.

- **app_pub_keys**

  The app_pub_keys list contains RSA public keys to allow encryption of cryptographic key material for storage at the provisioning server and eventual import in the network backend. If the host does not require cryptographic keying, this can be ignored.

- **app_files**

  The app_files list contains objects of type npt_comm.TargetFileInfo(). Each of these objects has three validated paths to files as listed below. The npt_comm code has already checked for access permissions and validated an md5 hash of each file before returning. The validated paths to files can be accessed as:

  1. cfg.app_files[0…].firmware_path: This is a firmware image.
  2. cfg.app_files[0…].field_config_path: This is a field configuration file.
  3. cfg.app_files[0…].factory_config_path: This is a factory configuration file.

The three files listed above may be the same if no separate configuration is necessary at test time. Given this information, the details of provisioning the host are up to the end device designer. After the host is successfully updated, configured, and any key information set, the tool may optionally send data to the provisioning server for storage and later use by the target network backend application. If there is no need for provisioning data to go to the backend network application, this step can be skipped.

C.5 Saving Host Provisioning Outputs to the Server

Data is sent to the server using the previous prov_srv object by calling prov_srv.setDutParams().This interface takes a number of parameters as indicated in the following table.

Table 7. List of Interface Parameters

Parameter	Description
`timestamp`	Number of seconds, in integers, since epoch. The timestamp can be generated by calling `int(time.time())` after importing the Python standard library module time.
`config_id` (or `network_id` in some provisioning releases)	The unique 64-bit ID used in the `getConfig()` call above.
`batch`	The batch number. For more information, see section 7.2.4.
`node_id`	A 32-bit integer MAC address for the node. The `node_id` can be set to "None," if not known, but can be read, for example, from a barcode scanner of the node label or from the node using `node_utils.py`.
`app_names`	The same list from the config response, can pass `cfg.app_names`.
`app_ids`	List (`len(app_names)` length) of a unique identifier for the end device, formatted as a hex string. List entry may be "None" or an empty string, if not applicable.
`app_keys`	List (`len(app_names)` length) of encrypted cryptographic information, typically encrypted with the appropriate `app_pub_keys`. RSA key (`npt_comm.EncryptKeyInfo`) provides a sample interface for such encrypted data which is formatted as a hex string. List entry may be "None" or an empty string, if not applicable.
`extraAppInfo`	List (`len(app_names)` length) of arbitrary application-specific information, which is formatted as a hex string. List entry may be "None" or an empty string, if not applicable.

The information reported to the provisioning server is exported in the key output file generated as discussed in section 7.3 Exporting Key File for Devices and can be ingested as needed by the application-specific backend network software.

Appendix D eMCM Provisioning

eMCM provisioning is an extension of host provisioning, as described in Appendix C of this guide. In addition to node provisioning, eMCM provisioning performs firmware updates, configuration, and key generation for the eMCM host processor. Optionally, eMCM provisioning may include an over-the-air test verifying connectivity to an On-Ramp Wireless communications system installed in the factory.

D.1 Installation

The eMCM provisioning client software consists of command line Windows executables. If a single-stage provisioning process is used, the appropriate executable is “emcm_provision.exe.” For multi-stage provisioning, the appropriate executable is “emcm_factory.exe.” In either case, the executable and all supporting files are distributed as a compressed tar archive and can be installed onto the PC that communicates with the eMCM by unpacking this archive into the operator’s desired location. Additional details on single and multi-stage provisioning are provided in section D.2.

For the most reliable results, the Windows driver settings for the COMM port(s) used to communicate with the eMCM should be adjusted for minimum latency. This may take different forms for different serial devices but generally the driver configuration should use:

- Minimum packet sizes,
- Minimum queue depths to generate an interrupt, and
- Latency timers should be set to minimum values (if applicable)

D.2 Operation

In some cases, the operator may desire to provision an eMCM in a single step (i.e., single-stage provisioning) resulting in a final field configuration. This precludes an over-the-air test during the provisioning process.

In other cases, the operator may wish to provision the eMCM in several steps (i.e., multi-stage provisioning) using emcm_factory.exe. A few reasons for provisioning an eMCM in several steps are:

- To perform an over-the-air test with a different node configuration than the target network configuration
- To force the eMCM into a quiet mode during meter calibration and test
- If after assembling an eMCM into a meter, the communication to the eMCM is slow, high message volume operations (particularly firmware updates) may need to occur in a fixture prior to assembling the eMCM into a meter.

D.2.1 Single-stage Provisioning Using emcm_provision.exe

In the case of single-stage provisioning, the operator (or a higher layer factory automation software) runs the "emcm_provision.exe" executable. For up-to-date usage instructions, running the program with a "-h" option displays all available program parameters. In general, these program parameters group into the options described in the following tables. A sample command line using many of the command options is provided below:

```
emcm_provision.exe -d COM3 -s 192.168.2.35 -B 1000 --config-alias=kite_170_node_637_dev_net
```

Communication Options

The following table provides communication options.

Table 8. Communication Options for Single-stage Provisioning

Command Option	Description
`-d <serial_port>, --device=<serial_port>`	The communication port to use with the eMCM.
`-b <BAUD>, --baud=<BAUD>`	The serial baud rate (default is 115200).
`-wrapper=<WRAPPER_TYPE>`	Apply protocol wrapper to serial in/out.
`-s <ip_addr>[:<port>], --server=<ip_addr>[:<port>], --prov_srvr=<ip_addr>[:<port>]`	The IP address or hostname of the provisioning server. The TCP port number to connect with the provisioning server.

Final Configuration Options

The following table provides options for choosing the final configuration to apply.

Table 9. Final Configuration Options for Single-stage Provisioning

Command Option	Description
`--factory-config`	Allows configuration with factory versions of the eMCM and node configuration files.
`--skip-keys`	Skip key provisioning.
`--skip_lock`	Prevents securing the AHP and flash/JTAG interfaces.
`-p <prov_config_id>, --config-id=<prov_config_id>`	The unique ID for the provisioning configuration—a 64-bit integer that defines the target network, application, and configuration.
`--config-alias=<alias>`	Allows use of a string alias for config ID rather than a numeric ID.

Diagnostic Output and Logging Options

The following table provides options for controlling diagnostic output and logging.

Table 10. Diagnostic Output and Logging Options for Single-stage Provisioning

Command Option	Description
`--verbosity=<0-2>`	■ 0 prints critical errors only to stdout ■ 1 prints warning and above to stdout (default) ■ 2 prints debug and above to stdout

Command Option	Description
`--logfile=<logname>`	Writes diagnostics to the specified log file.

Miscellaneous Options

Table 11. Miscellaneous Options for Single-stage Provisioning

Command Option	Description
`-B <batch>, --batch=<batch>`	The batch number for the current provisioning run. The batch number is an arbitrary integer number that can be assigned to each provisioning run of one or more nodes for tracking purposes. The batch number is saved in the provisioning server key database and can be used to select keys for key export.
`--path_trans_from=<string>, --path_trans_to=<string>`	Allows re-writing of base path in files referenced in xml from server. For example: `--path_trans_from=/workspace --path_trans_to=z:` might allow sharing a single server between windows and Linux clients.
`--partial_mcm_id=<partial_mcm_id>`	The last 8 characters of the eMCM ID is expected to be a unique 8-digit hex identifier which is checked against the eMCM ID reported by the eMCM.

D.2.2 Multi-stage Provisioning Using emcm_factory.exe

In this case, the operator (or higher layer factory automation software) runs the "emcm_factory.exe" executable several times, specifying the operation to run. Available operations are described in the following table.

Table 12. Available Operations for Multi-stage Provisioning

Operation	Description
PRETEST	This operation performs any required firmware updates, configures the eMCM and node for the factory On-Ramp Wireless communications system, and clears the node status from the communication system gateway (ensuring that an up-to-date status is found by the POSTTEST operation).
CHECKCONFIG	This operation is optional and simply ensures that eMCM communication is passing and that an appropriate PRETEST operation has been run. This is useful if the PRETEST is run separately from the POSTTEST. For example, in the case where the PRETEST is performed at board-level in a fixture and POSTTEST is performed after assembly into a meter.

Operation	Description
POSTTEST	This operation does the following: ■ Checks to see that the node has successfully joined the factory network ■ Applies final field configurations to the eMCM and node ■ Generates and applies security keys to the eMCM and node ■ Verifies that the meter-to-eMCM communications link is valid.

For up-to-date usage instructions, running the program with a "-h" option displays all available program parameters. In general, these program parameters group into the options described in the following tables.

Communication Options

The following table provides communication options.

Table 13. Communication Options for Multi-stage Provisioning

Command Option	Description
`-d <serial_port>, --device=<serial_port>`	The communication port to use with the eMCM.
`-b <BAUD>, --baud=<BAUD>`	The serial baud rate (default is 115200).
`-wrapper=<WRAPPER_TYPE>`	Apply protocol wrapper to serial in/out.
`-s <ip_addr>[:<port>], --server=<ip_addr>[:<port>], --prov_srvr=<ip_addr>[:<port>]`	The IP address or hostname of the provisioning server. The TCP port number to connect with the provisioning server.
`-g <Gateway Address> --gateway=<Gateway Address>`	Specifies the Gateway IP Address. The default is localhost.
`--control_port=<Gateway Control Port Number>`	Specifies the Gateway control port number. The default is 4021.
`--csys_1_x`	Allows operation with a 1.x gateway for OTA test purposes. The default is 2.x.

Final Configuration Options

The following table provides options for choosing the final configuration to apply.

Table 14. Final Configuration Options for Multi-stage Provisioning

Command Option	Description
`--skip_lock`	Prevents securing the AHP and flash/JTAG interfaces.
`-p <prov_config_id>, --config-id=<prov_config_id>`	The unique ID for the provisioning configuration—a 64-bit integer that defines the target network, application, and configuration.
`--config-alias=<alias>`	Allows use of a string alias for config ID rather than a numeric ID.

Diagnostic Output and Logging Options

The following table provides options for controlling diagnostic output and logging.

Table 15. Diagnostic Output and Logging Options for Multi-stage Provisioning

Command Option	Description
`--verbosity=<0-2>`	■ 0 prints critical errors only to stdout ■ 1 prints warning and above to stdout (default) ■ 2 prints debug and above to stdout
`--logfile=<logname>`	Writes diagnostics to the specified log file.

Miscellaneous Options

Table 16. Miscellaneous Options for Multi-stage Provisioning

Command Option	Description
`-B <batch>, --batch=<batch>`	The batch number for the current provisioning run. The batch number is an arbitrary integer number that can be assigned to each provisioning run of one or more nodes for tracking purposes. The batch number is saved in the provisioning server key database and can be used to select keys for key export.
`--path_trans_from=<string>, --path_trans_to=<string>`	Allows re-writing of base path in files referenced in xml from server. For example: `--path_trans_from=/workspace --path_trans_to=z:` might allow sharing a single server between windows and Linux clients.
`--partial_mcm_id=<partial_mcm_id>`	The last 8 characters of the eMCM ID is expected to be a unique 8-digit hex identifier which is checked against the eMCM ID reported by the eMCM.
`-t <Timeout Value> --timeout=<Timeout Value>`	Specifies the OTA test timeout value in seconds. The default is 600 seconds.

A sample series of command lines for using the “emcm_factory.exe” program are provided below:

```
emcm_factory.exe -d COM3 -s 192.168.2.35 -g 192.168.6.77 -B 1000 --
config-alias=kite_170_node_637_dev_net --logfile=emcm_factory_s1.log
PRETEST

emcm_factory.exe -d COM3 -s 192.168.2.35 -g 192.168.6.77 -B 1000 --
config-alias=kite_170_node_637_dev_net --logfile=emcm_factory_s2.log
POSTTEST
```

See usage examples for specific meters for further samples.

D.3 Meter-specific Usage and Factory Procedures

i210+, i210+c and SGM3xxx

These meters allow for direct serial access to the eMCM while assembled, so the complete factory flow is performed at one station (WECO fixture) after meter assembly. The test process consists of running a PRETEST operation before meter calibration & test, then running meter calibration and test, followed by running a POSTTEST operation.

Sample command lines are shown below:

```
emcm_factory.exe -d COM3 -s 192.168.2.35 -g 192.168.6.77 -B 1000 --
config-alias=82JX570141 PRETEST

<WECO calibration and test>

emcm_factory.exe -d COM3 -s 192.168.2.35 -g 192.168.6.77 -B 1000 --
config-alias=82JX570141 POSTTEST
```

kv2c

These meters do not allow for serial access to the eMCM, only (much slower and higher bit error rate) optical port messaging. Therefore, the message intensive PRETEST operation is performed at eMCM board level in a test fixture. The boards are then assembled into meters, and the rest of the test sequence is performed on a WECO test station.

On a fixture, the PRETEST step is performed as usual. This is followed by meter assembly. Since this is potentially error prone and the meter calibration and test is fairly lengthy, it is advantageous to perform a sanity check of the eMCM basic functionality and configuration before starting meter calibration and test. This is performed with a CHECKCONFIG procedure, then as normal the WECO calibration and test and POSTTEST steps follow.

For the CHECKCONFIG and POSTTEST steps, note that the optical port communication link requires extra option flags. In particular, it runs at 9600 baud, and has extra headers on messages, requiring a wrapper. This is accomplished by passing the extra options shown below to the provisioning executable:

```
-b 9600 --wrapper=opt_psem
```

Sample command lines are shown below:

```
emcm_factory.exe -d COM3 -s 192.168.2.35 -g 192.168.6.77 -B 1000 --
config-alias=82JX570141 PRETEST

<Meter/EMCM assembly, move to WECO station>

emcm_factory.exe -d COM20 -b 9600 --wrapper=opt_psem -s 192.168.2.35 -g
192.168.6.77 -B 1000 --config-alias=82JX570141 CHECKCONFIG

<WECO calibration and test>

emcm_factory.exe -d COM20 -b 9600 --wrapper=opt_psem -s 192.168.2.35 -g
192.168.6.77 -B 1000 --config-alias=82JX570141 POSTTEST
```

Appendix E Abbreviations and Terms

Abbreviation/Term	Definition
3DES	Triple Data Encryption Standard
ACK	Acknowledgement
AES	Advanced Encryption Standard
AMI	Advanced Metering Infrastructure
AP	Access Point
CA	Certificate Authority
CDLD	Code Download
CMAC	Cipher-based Message Authentication Code
CRC	Cyclic Redundancy Check
CSR	Certificate Signing Request
CSV	Comma Separated Values
CTS	Clear to Send
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DOS	Disk Operating System
DUT	Device Under Test
FQDN	Fully Qualified Domain Name
FTP	File Transfer Protocol
GW	Gateway
HES	Head End System
HTTP	HyperText Transport Protocol
KGF	Key Generation Function
KMS	Key Management Server
KPA	Key Provisioning Agent
Node	The wireless module developed by On-Ramp Wireless that integrates with OEM sensors and communicates sensor data to an Access Point. Also, the generic term used interchangeably with end point device.
OTA	Over-the-Air
PEM	Privacy-Enhanced Mail
RSA	Rivest, Shamir, Adleman encryption algorithm.
RTS	Ready to Send
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UNIL	Universal Node Interface Library