SmartMesh IP Application Notes

Table of Contents

1 Revision History

Revision	Date	Description
1	03/18/2013	Initial Release
2	07/18/2013	Added "What is Packet ID and why do I Need it?" Application Note
3	10/22/2013	Minor corrections
4	01/30/2014	Added "Overlapping Networks" Application Note; Modified "Building a Mesh-of-Meshes" Application Note to reflect sample implementation
5	04/04/2014	Clarified how Q is calculated
6	04/30/2014	Updated "Building Deep IP Networks" Application Note
7	10/28/2014	Minor changes to "SmartMesh Security" Application Note; Clarified boot event flags in "Data Publishing for SmartMesh IP" Application Note; Clarified need to connect APIExplorer to measure idle current in "How to Evaluate Network and Device Performance" Application Note
8	04/21/2015	Updated "Identifying and Mitigating the Effects of Interference", "Data Publishing for SmartMesh IP", and "How to Evaluate Network and Device Performance" Application Notes
9	12/03/2015	Added "How to Read Mote Parameters Remotely" Application Note; Updated hardware descriptions
10	05/25/2016	Added "6LoWPAN and Routing in a SmartMesh IP Network" and "Building an OTAP Application" Application Notes
11	07/08/2016	Renamed "Predicting Network Health" Application Note to "Predicting Embedded Manager Network Health Using the CLI" and added examples
12	07/28/2016	Updated Power section of "How to Evaluate Network and Device Performance" Application Note
13	09/19/2016	Added "Monitoring Background Noise" Application Note
14	11/01/2016	Removed several outdated Application Notes; Renamed and revised several Application Notes to distinguish between Embedded Manager and VManager deployments
15	01/30/2017	Added Related Documents page

2 Application Note Summary

This document contains a collection of Application Notes about the SmartMesh IP product family:

- How to Evaluate Network and Device Performance - Methods for measuring a host of performance metrics.
- How to use the Embedded Manager Subscribe API Filters - Strategies for monitoring manager notifications.
- Monitoring SmartMesh IP Network Health with the Embedded Manager - Automated checks to ensure that your network is performing well.
- Data Publishing for SmartMesh IP - Low-level descriptions of how to use the mote's API for sending data.
- Managing Advertising in an Embedded Manager Network - A state machine to save power by turning advertising off.
- Improving Latency with the Embedded Manager Powered Backbone - Description of several backbone use cases.
- Building Deep IP Networks - Settings to use for networks up to 32 hops deep.
- Overlapping Networks - Calculations of how many motes can safely coexist in the same radio space.
- How to Read Mote Parameters Remotely - Explains using the netGetParameter command to query mote parameters over the air.
- 6LoWPAN and Routing in a SmartMesh IP Network - Description of the IPv6-related headers as used in a network.
- Building a VManager OTAP Application - Guide for implementing Over-The-Air-Programming to update mote code using the VManager.
- Building an Embedded Manager OTAP Application - Guide for implementing Over-The-Air-Programming to update mote code using the Embedded Manager.
- Predicting Embedded Manager Network Health Using the CLI - Initial post-deployment assessment tips.
- Monitoring Background Noise - Using the channel-specific health reports to diagnose wireless interference.

The following Application Notes apply to both SmartMesh IP and WirelessHART families. Differences between the products, if any, are highlighted:

- Planning A Deployment - High-level considerations prior to deploying a network.
- Common Problems and Solutions - General troubleshooting advice.
- Changing Provisioning Factor to Increase Manager Throughput - Decreasing per-mote bandwidth to support more total traffic in high-stability conditions.
- Debugging Congested Networks - Understanding impact on network operation when mote queues are full and tips on remedying such situations.
- Identifying and Mitigating the Effects of Interference - Graphically representing network statistics to notice interference-related problems.
- Obtaining Accurate Timestamps - How to use the network time synchronization properly for timestamping packets.
- Using Multiple Managers to Build Large Networks - Considerations for deployments exceeding the mote limits of a single manager.
- Using the SmartMesh Power and Performance Estimator - Examples on using the spreadsheet for predicting power and latency in a variety of networks.
- What to Expect with Motes That Move - Details on the behavior of a mote moving to a different location in the same network.

- Moving Motes Between Networks - Details on how to migrate motes between different networks.
- Configuring a Network for Bounded Data Latency - Adding bandwidth to a network to decrease the packet latency.
- Network Coexistence - Features that allow multiple networks to operate in the same radio space.
- How to Choose a Join Duty Cycle - Considerations for trading off power and join speed.
- SmartMesh Security - A description of the security used in all SmartMesh networks.
- Using the TestRadio Commands - A description of how to use APIs for testing radio performance or certification.
- Best Practices to Limit Average Current During Peak Periods - Guidelines to follow in order to keep average current down when booting or writing flash.
- Methodology For Pilot Network Evaluation - Overview of deployment and statistics collection for network pilots.
- What is Packet ID and why do I Need it? - Details on the single bit used to provide reliable call and response with the Sensor Processor.

3 Related Documents

The following documents are available for the SmartMesh IP network:

Getting Started with a Starter Kit

- SmartMesh IP Easy Start Guide - walks you through basic installation and a few tests to make sure your network is working
- SmartMesh IP Tools Guide - the Installation section contains instructions for installing the serial drivers and example programs used in the Easy Start Guide and other tutorials.

User's Guide

- SmartMesh IP User's Guide - describes network concepts, and discusses how to drive mote and manager APIs to perform specific tasks, e.g. to send data or collect statistics. This document provides context for the API guides.

Interfaces for Interaction with a Device

- SmartMesh IP Manager CLI Guide - used for human interaction with a Manager (e.g. during development of a client, or for troubleshooting). This document covers connecting to the CLI and its command set.
- SmartMesh IP Manager API Guide - used for programmatic interaction with a manager. This document covers connecting to the API and its command set.
- SmartMesh IP Mote CLI Guide - used for human interaction with a mote (e.g. during development of a sensor application, or for troubleshooting). This document covers connecting to the CLI and its command set.
- SmartMesh IP Mote API Guide - used for programmatic interaction with a mote. This document covers connecting to the API and its command set.

Software Development Tools

- SmartMesh IP Tools Guide - describes the various evaluation and development support tools included in the SmartMesh SDK , including tools for exercising mote and manager APIs and visualizing the network.

Application Notes

- SmartMesh IP Application Notes - Cover a wide range of topics specific to SmartMesh IP networks and topics that apply to SmartMesh networks in general.

Documents Useful When Starting a New Design

- The Datasheet for the LTC5800-IPM SoC, or one of the modules based on it.
- The Datasheet for the LTC5800-IPR SoC, or one of the embedded managers based on it.
- A Hardware Integration Guide for the mote/manager SoC or module - this discusses best practices for integrating the SoC or module into your design.

- A Hardware Integration Guide for the embedded manager - this discusses best practices for integrating the embedded manager into your design.
- A Board Specific Integration Guide - For SoC motes and Managers. Discusses how to set default IO configuration and crystal calibration information via a "fuse table".
- Hardware Integration Application Notes - contains an SoC design checklist, antenna selection guide, etc.
- The ESP Programmer Guide - a guide to the DC9010 Programmer Board and ESP software used to load firmware on a device.
- ESP software - used to program firmware images onto a mote or module.
- Fuse Table software - used to construct the fuse table as discussed in the Board Specific Configuration Guide.

Other Useful Documents

- A glossary of wireless networking terms used in SmartMesh documentation can be found in the SmartMesh IP User's Guide
- A list of Frequently Asked Questions

4 Application Note: How to Evaluate Network and Device Performance

4.1 Introduction

OK, you just got a SmartMesh IP starter kit - now what? This app note describes a number of tests you can run to evaluate specific performance characteristics of a SmartMesh network using the Embedded Manager. It assumes that you have installed the SmartMesh IP SDK python tools, and the necessary FTDI drivers to communicate with manager and motes. It also presumes you have access to the Manager and Mote CLI and API documentation.

First thing is to connect to the manager and log into the manager's CLI as described in the Introduction in the SmartMesh IP Manager CLI Guide

```
> login user
```

You will also need to connect a DC9006 Eterna Interface Card to one or more DC9003A-B motes in order to access the mote's CLI to configure it.

4.2 Join Behavior

How fast does a mote join? Mote join speed is a function of 6 things:

- Advertising rate - the rate at which motes in-network advertise. The user has little control of this other than mote density, or turning it off explicitly at the manager
- Join duty cycle - how much time a searching mote spends listening for a network versus sleeping
- Mote join state machine timeouts - there is no user control over these timeouts in SmartMesh IP
- Downstream bandwidth - affects how quickly motes can move from being synchronized to the network to **Operational**, i.e. able to send data
- Number of motes - contention among many motes simultaneously trying to join for limited resources slows down joining with collisions
- Path stability - over which the user has little or no control; path stability is the ratio of successful (acknowledged) packets to sent packets between a pair of motes

The join process is broken up into three phases: search, synchronization, and message exchange.

- Search time is determined by advertising rate, path stability, and mote join duty cycle - this can be 10's of seconds to 10's of minutes, largely depending upon join duty cycle
- Synchronization is determined by mote state machine timeouts - this period is only a few seconds in SmartMesh IP.

- Message exchange is determined by downstream bandwidth and number of motes (which compete for downstream bandwidth). This is a minimum of ~10 s per mote, and can be much slower.

So, for example, it may seem that to join a network quickly, you would set the join duty cycle to maximum. However, this may result in a lot of motes competing for downstream bandwidth, and a network may form more quickly with a lower setting. In the experiments below, you will examine the "knobs" you have to control search and message exchange phases.

4.2.1 Tradeoff Between Search Time and Average Current

An unsynchronized mote listens for manager and other mote advertisements to synchronize to the network. The fraction of time spent listening versus sleeping is called the *join duty cycle*. The join duty cycle is one-byte field that can be set from 0 (0.2%) to 255 (almost 100%) - the default for Starter Kits is 100% or a setting of 255. Lower settings result in longer search times at lower average current, while higher settings shorten search but increase average current. Provided that at least one mote is advertising in the mote's vicinity, the **energy** dedicated to joining remains the same - join duty cycle only affects average **current**. If a mote is isolated, or if the manager is not on, or if advertising has been turned off, a mote could potentially exhaust its battery looking for a network. Join duty cycle gives the user control over the tradeoff between speed when a network is present and how much energy is used when it isn't.

The time for a mote to synchronize and send in its join request is the first visible sign of a mote joining a network. You can activate the manager's mote state trace using the `trace` CLI command:

```
> trace motest on
```

When the manager sees the mote's join request, it will mark the mote as being in the **Negotiation1** (Negot1) state.

To see the effect of join duty cycle:

- Connect to the manager CLI and turn on the mote state trace as described above.
- Connect to the mote CLI by using an Eterna Interface Card (DC9006)
- Reset the mote - it will print out a reset message

```
> reset
SmartMesh IP mote, ver 1.1.0.32 (0x0)
```

this will cause the mote to begin searching for the network with it's default duty cycle. You should see a series of notifications as it changes state, with a timestamp in ms followed by the state:

```
52727 : Joining
56227 : Connected
60121 : Active
```

It should take <10 seconds on average for a mote to synchronize and sent its join request. You may want to repeat several times (resetting each time) to see the distribution of synchronization times.

- Reset the mote and use the mote CLI to set the join duty cycle to 5% (0.05 * 255 = 13) - this is controlled through the mote `mset joindc` CLI command Join duty cycle is non-volatile in a SmartMesh IP mote - it persists through reset or power cycle, so you only need to do this step once for each join duty cycle change. Measure how long it takes the mote to transition to the **Negotiating1** state, repeating the measurement as described above. It should be ~3 min on average.

```
> mset joindc 13
```

- Reset the mote and use the mote CLI to set the join duty cycle to 25% (64). Measure how long it takes the mote to transition to the **Negotiating1** state. It should be ~30 s on average.

```
> mset joindc 64
```

Note: When the manager sees a join request from a mote that was previously operational, it will mark it **Lost** first before promoting it to connected. It will appear as:

```
264504 : Mote #2 State:   Lost -> Negot1
266178 : Mote #2 State: Negot1 -> Negot2
```

4.2.2 Measuring Join Time

It takes a number of packets sent by the manager and acknowledged by the mote to transition a mote to the **Operational** (Oper) state where it can begin sending data. We can use the manager mote state trace CLI command (this trace should still be on from the test above) to see these state transitions and measure how long they take. The timestamps are in ms. Reset the mote and watch it transition to **Operational**. Each line in the following trace represents one handshake between the mote and the manager:

```
117757 : Created mote 2
117765 : Mote #2 State:   Idle -> Negot1

119853 : Mote #2 State: Negot1 -> Negot2

122366 : Mote #2 State: Negot2 ->  Conn1

125222 : Mote #2 State:  Conn1 ->  Conn2

127127 : Mote #2 State:  Conn2 ->  Conn3

129185 : Mote #2 State:  Conn3 ->   Oper
```

Here it took ~11.5 s to transition.

4.2.3 Effect of Downstream Bandwidth on Join

The rate at which the manager can send packets is a function of the downstream slotframe size - the manager assigns the same number of dowstream links, regardless of slotframe size, so a 4x longer slotframe has 1/4 the bandwidth. This is shown in Figure 1 - Slot 1 in a 100-slot slotframe repeats twice as often as the same slot in a 200-slot slotframe. By default, the manager builds the network on a "fast" 256-slot slotframe, and then transitions to a "slow" slotframe after one hour. The slow slotframe can be 256 (default), 512, or 1024 slots - this is set via the *dnfr_mult* (downstream slotframe multiplier) parameter. A longer frame means fewer downstream listens per unit time, so it will lower the average current for all motes, but also slow down the join process (and any other downstream data) for motes added later. While this has no effect on the time it takes to synchronize to the network (time to the **Negotiating1** state), it spaces out the other state transitions.

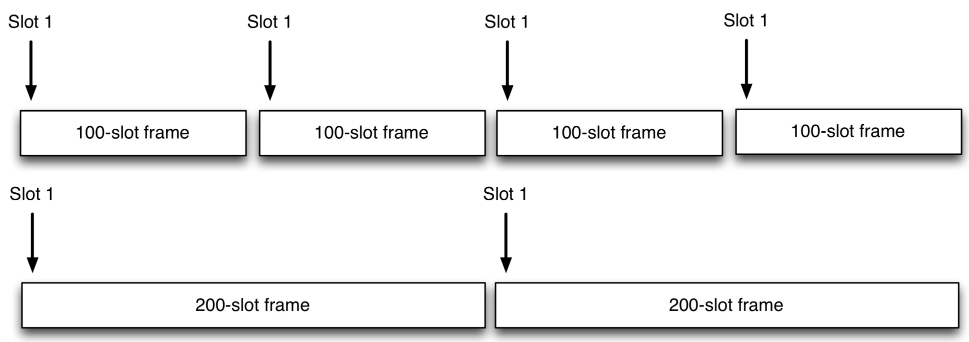


Figure 1 - Slot 1 repeats more often on a 100-slot frame than in a 200-slot frame

Note these are the nominal sizes - the actual slotframe sizes are randomized a bit to improve coexistence when multiple networks are overlapping. We will refer to their nominal sizes throughout.

To see the effect of longer downstream slotframe:

- Via manager CLI, confirm that the normal "fast" slotframes are being used. Look for the line that shows the downstream slotframe multiplier (*dnfr_mult*):

```
> show config
netid = 1229
txpower = 8
frprofile = 1
maxmotes = 17
basebw = 9000
dnfr_mult = 1
numparents = 2
cca = 0
channellist = 00:00:7f:ff
autostart = 1
locmode = 0
bbmode = 0
bbsize = 1
license = 00:00:00:00:00:00:00:00:00:00:00:00:00
ip6prefix = fe:80:00:00:00:00:00:00:00:00:00:00:00:00:00:00
ip6mask = ff:ff:ff:ff:ff:ff:ff:ff:00:00:00:00:00:00:00:00
radiotest = 0
bwmult = 300
```

- Now you will adjust the downstream slotframe multiplier to give a 1024 slot slotframe - by setting the multiplier to 4, you get a 4*256 slot slotframe. Config parameters are non-volatile:

```
> set config dnfr_mult 4
```

- Force the longer slotframe use:

```
> exec setDnframe normal
Start Global Unicast Command setDnFrame
```

This may take a couple minutes to take effect.

- Confirm that the manager is using the longer downstream slotframe:

```
> show status

S/N: 00-17-0D-00-00-38-0C-8C
MAC: 00-17-0D-00-00-38-0C-8C
IP6: FE:80:00:00:00:00:00:00:00:17:0D:00:00:38:0C:8C
HW ver: 1.2
NetworkID: 293
Base bandwidth (min links): 9000 (1)
Frame size Up/Down 277/1052. Number of working timeslots 256.
Available channels 15.
Base timeslot(TS0) 20
Network mode is Steady State
Downstream frame 1052 timeslots
Optimization is On
Advertisement is On
AP output CTS is Ready
Backbone is off
Location is off
API is Not Connected
License 00:00:00:00:00:00:00:00:00:00:00:00:00
Network Regular Mode
Total saved links 3
```

Manager CLI commands often use the term "frame" instead of the more formal "slotframe." "Frame" is never used to indicate an 802.15.4 packet.

- Now power cycle the single mote again and note the time it takes for the mote to transition from **Negotiating1** to **Operational**. This should be ~4x the "fast" join time you saw above.

Path stability also affects join time, as worse path stability means more retries. Having a mesh architecture means that the effect of any individual path is minimized.

4.2.4 Network Formation vs Single Mote Join

The number of motes joining simultaneously also affects network formation time, as these motes must compete for limited join links and downstream bandwidth. To see this effect (this assumes you are using the longer downstream frame from above):

1. Configure all of your motes for 100% join duty cycle (as described above) - we aren't concerned with average current here
2. Reset a single mote, and let it join. Repeat this 10 times to get an average join time.
3. Reset all motes and note the time it takes for the network to form completely, i.e. all motes transition to **Operational** when the join trace is on. You may want to repeat this experiment a few times to get a feel for the variability.

Having 5 motes means, on average, that someone will hear the advertisement faster than one mote would in a single mote network so you may see the first mote join quicker in the 5-mote case. However, if too many motes synch up at once they will contend for the same shared Access Point (AP) links which can slow down the overall network join time.

Return the manager to the "fast" downstream frame before proceeding:

```
> set config dnfr_mult 1
```

- Reset the manager and log back in.

4.2.5 Measuring Time to Recover from a Lost Mote

One of the reasons a mesh is important is the robustness to path failures. In star and tree structures, a single RF path can represent a single point of failure for one or many devices data. In evaluating the mesh, many observers will want to see the mesh recover from a path failure. The most reliable way to make a path fail is to power off a device. Many observers also care a great deal about recovery time after the loss of a device, so this test can address both.

First we must decide what we mean by 'recovery'. If a user walks up to a 10-device mesh, and powers down one device, we consider the network recovered if the only change in data delivery is that the one node powered down stops sending data. If all other nodes continue to send data at their configured rates without interruption, then we consider the recovery time to be zero. The network survived the loss of a node with no disturbance to the remainder of the network. Another way to look at it is to establish some metric for quality of service, like data latency, that is being met prior to powering down a node, and then looking for the loss of the node to cause that QoS metric to degrade, and look for that QoS metric to get back to where it was before. This involves more of a judgement call from the observer and depending on the reporting rates might not be easy to assign a hard time value. The third way to identify recovery is to power down a node that is a parent of one or more nodes, and measure the time it takes for the network to detect that a parent has been lost and establish a new parent.

To summarize the three different "time to recover" test motivations are:

1. Uninterrupted data delivery. If powering down a mote causes no interruption to data delivery from any other motes, then time to recover = 0. If data delivery is interrupted, then time to recover is the time until data delivery from all nodes its restored
2. QoS data delivery. Baseline the QoS of a network through data analysis. Power down a node, and note degradation in the QoS metric for some or all nodes. Time to recover is the time until that previous QoS metric is met again
3. Time to repair the mesh. Look at the mesh. Power down a node that will cause other nodes to have only a single parent. Time to recover is the time it takes for the network to detect the loss and re-establish a full mesh

Test 1 and 2 are essentially the same. We recommend you deploy a network (see Tip below) and connect to CLI and turn `trace stat on`. Each packet received by the manager will then print the Mote ID and the latency of that packet. Capture this text for some time, and then power down a mote. The mote you power down could be chosen randomly or you could specifically identify a mote with many children in the mesh. Note an approximate timestamp in your text capture when you powered down a node. Let the network continue to run for several minutes. Then you should be able to plot the data from this text capture to identify the moment when the last packet was received from the powered down mote, and after that time you should be able to identify if there are any gaps or delays in data delivery from any other nodes. You will need to provide your own parsing an data analysis tools - MATLAB, Excel, and Python are all suitable.

✓ Wait for the first few motes to join before powering up the remaining motes to form a mesh more quickly than if you power up all motes simultaneously. Motes report who they heard advertising when they join - when you power up all motes at the same time, only the AP is advertising, so likely motes will only report the AP as a possible neighbor. Motes discover and report other neighbors slowly to minimize energy, it takes up to 5 minutes after joining for the manager to begin "meshing" the network. In real deployments the delay to discover neighbors is unimportant, but in demo or evaluation scenarios, waiting for a few motes to join allows immediate meshing.

Test 3 involves a different trace function on the manager. Deploy your initial network, and observe that a mesh has been established (i.e. if numparents=2, then all motes will have two parents except one mote will have exactly one parent). Start a text capture of the manager `trace link on`. Select and power down a mote. Note the approximate timestamp for the moment you powered the mote down and watch the events associated with detecting paths that have failed and the activity associated with establishing new paths. After a few minutes stop the trace, and convince yourself that a full mesh has been restored. The 'time to recover' is the time from when you powered down the mote to the timestamp of the final link add event in your capture.Expected results

1. If the network was a good mesh to start with, we expect no data loss and no additional mote loss due to the removal of a single device.
2. At moderate report rates (one packet per 5s or slower), we expect no disturbance in QoS. At faster report rates, you may see latency delays associated with a single device loss. Motes that had ~200ms latency might have ~500ms latency for a short time. Recovery time should be between one and two minutes.
3. Powering down a mote should be detected and repaired between one and two minutes

✓ The Stargazer GUI tool, which is typically installed with an evaluation kit, allows you to see the mesh forming

4.3 Latency

4.3.1 Measuring Latency

Activating the manager CLI statistics trace provides a latency measure for every upstream packet received at the manager:

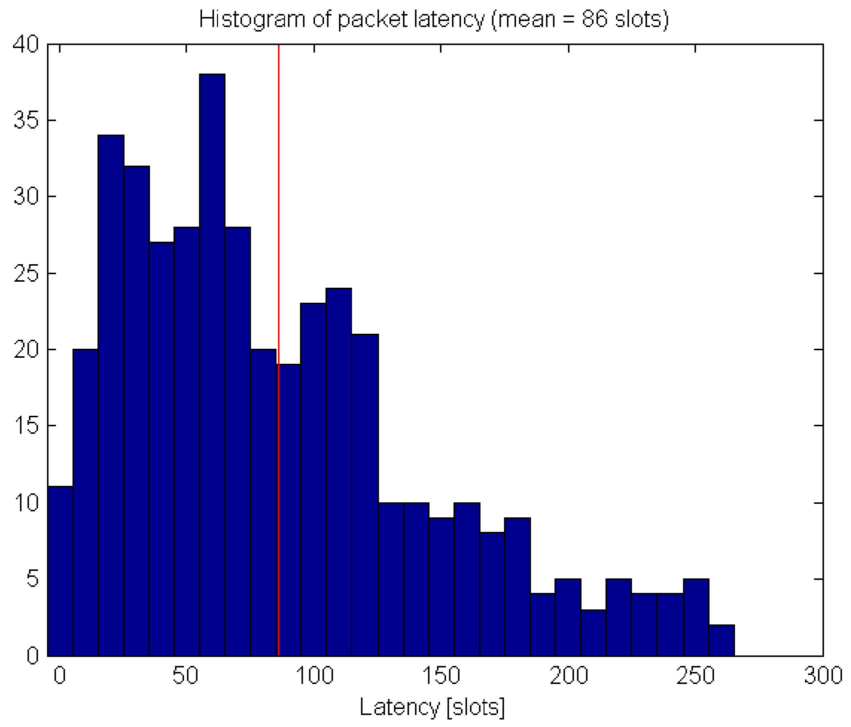
```
> trace stats on

281098 : STAT: #2: ASN Rx/Tx = 38694/38570, latency = 899, hops = 1
281100 : STAT: new average hops10=10 latency/10=56
```

This trace has two lines of report for each upstream packet. The first line shows when, in Absolute Slot Number (ASN) the packet was received (Rx) at the manager and when the packet was generated (Tx) at the mote. ASN is the number of timeslots that have elapsed since the network was started, or 20:00:00 UTC July 2,2002 if manager UTC time is set. The difference between these two counts, multiplied by 7.25 ms per slot, gives the value reported here as 899 ms. The time between the sensor generating the data, and the notification at the manager will be slightly longer, as there may be some queueing delay. The first line also shows how many hops upstream the packet took.

The second line shows the manager averaging the statistics after receiving this packet. It shows the average hops for this mote (in units of 0.1 hops) and the average latency (in 10 ms units). So here the average latency is 56*10 ms = 560 ms.

You can collect a large number of latency trace prints and plot a distribution. It should look something like this:



On average, unless path stability is very low, we expect a per-hop upstream latency of < 1/2 the upstream slotframe length.

4.3.2 Comparison with a Star Topology

In ZigBee networks, sensor nodes typically report to powered, always-on routers. As a result, data latency can be very low - until the link fails.

Dust's networks offer a number of means to achieve low data latency - through a powered backbone (see note below) which affects the entire network, through network-wide base bandwidth settings, or through per-mote services.

Try the following experiment:

- Place the motes within a few meters of each other and form a mesh network (default).
    - Measure average latency as described in the previous section. By default each mote will send temperature data once every 30 s, so you should take > 10 minutes of data.
- Form a star network, by forcing the motes to be non-routing. This is done via the mote CLI*:*

```
> mset rtmode 1
> mset maxStCur 20
```

- *rtmode* and *maxStCur* are non-volatile - it persists through reset/power cycle, but it must be changed before the mote joins the network or the mote must rejoin the network if the setting is changed after joining. You will need to connect the DC9006 board, use CLI to change the routing mode, then disconnect the DC9006 and repeat with another mote.

- We use the *maxStCur* setting here to ensure that the mote is in the absolutely lowest power mode possible.
- Measure average latency - you should see it has decreased slightly, as now motes will only have the AP as a parent. This star configuration makes motes vulnerable to link failures, and didn't really improve latency much, and isn't really the right solution to improve latency.

- Set the motes back to routing-capable, and turn the upstream backbone on - this is done by the `set config` manager CLI command. The backbone is a short slotframe with contention-access slots (as opposed to the normal dedicated slots used in other frames) that can be used to provide low-latency shared bandwidth across many motes. To get the lowest possible power mote configuration, we'll also eliminate the manager downstream multicast links by setting *nummlinks* to zero. Note that you need to input the superuser password prior to making this change.

```
> set config bbsize 1
> set config bbmode 1
> su becareful
> seti ini nummlinks 0
```

- After this, a manager reset is required for the backbone to come on. Backbone mode is non-volatile and persists through reset.
- After reconnecting to the manager and joining all motes, turn manager CLI latency trace back on:

```
> trace stats on
```

- Measure the average latency - it should be dramatically lower, despite motes occasionally routing through peers.

- Turn off the backbone:

```
> set config bbmode 0
```

- Reset the manager and turn the stats trace back on.

- Increase the base bandwidth in the network. The default is 9000 ms (i.e. 1 packet per 9000 ms, or .11 packets/s). You will set it to 1000 ms. This is also done with the `set config` command:

```
> set config basebw 1000
```

- After this, a manager reset is required.
- After reconnecting to the manager and joining all motes, turn CLI latency trace back on
- Measure average latency - it should have dropped. Even though motes are not publishing any faster, they have received more links, so latency decreases.

- Set the base bandwidth back to 9000 ms.

```
> set config basebw 9000
```

- Request a service on one mote.
- Measure average latency - you should see that it dropped for the mote with a service, and may have dropped a little on other motes that have that mote as a parent.

> ⚠ By default, motes only have a transmit link in the upstream powered backbone - this only has a miniscule effect on mote power. In order to receive (and thus forward packets for other motes on the backbone) the *setParameter<pwrSource>* API must be invoked on the mote. This is discussed in the app note "Application Note: Improving Latency with the Embedded Manager Powered Backbone." This can improve latency network wide - either upstream, or bi-directionally, but it increases mote current considerably. However, even with a bidirectional backbone on, a Dust mote will still be 10x lower current than a typical ZigBee router.

4.3.3 Tradeoff Between Power and Latency

In general there is a tradeoff between power and latency. The more links a mote has, the lower the latency will be on any given packet, since we tend to space links relatively evenly throughout the slotframe. This means that motes that forward traffic have lower latencies than motes that don't, even if they generate packets at the same rate. It also means that a mote can improve latency by asking for services at a mean latency target even if the data generation rate is lower.

Some of the power measurement experiments in the Power section show the correlation between power and latency.

4.3.4 Roundtrip Latency

Dust's networks are designed to primarily act as data collection networks, so most bandwidth is spent on upstream traffic. While the use of services can improve the upstream contribution, if fast (< 2 s per message) call-response is needed, the bidirectional backbone can be used. Try the following experiment:

- Turn the `io` and `iodata` traces on via the manager CLI
- Use APIExplorer in manager mode to send acknowledged application packet to a mote. This is done using the *sendData* API and using the OAP protocol - set the source/destination ports= `61625` (0xF0B9), and the payload = `050002FF020302000101.` (you can turn it off again by changing the last byte to **00**). This turns the indicator LED on on the mote, and causes the mote to acknowledge that the LED has been set. You should see a print similar to the following on the manager CLI:

```
442887 : TX ie=32:2c mesh=1222 ttl=127 asn=59614 gr=2 dst=2 rt=r4 r=1:2 tr=u:req:8; sec=s nc=0
ie=fe:00 IP=7d77 UDP=f7 sP=f0b9 dP=f0b9;

443503 : TxDone

444269 : RX ie=32:1c mesh=4002 ttl=127 asn=59699 gr=1 src=2 rt=g tr=u:req:0; sec=s nc=11 ie=fe:00
IP=7d77 UDP=f7 sP=f0b9 dP=f0b9;
```

Note that the 2nd byte in the payload is a sequence number - it must be incremented appropriately if multiple OAP commands are issued as discussed in the SmartMesh IP Tools Guide section on the On-chip application protocol.

Certain traces can generate more output than can be processed at the default CLI 9600 baud speed. In order to enable these traces you will first need to enable high-speed CLI prints:

> su becareful

> debug hscli on

Note that this does not change the CLI speed - it only enables the suppressed traces. You should only use this command for testing in small networks, as it can impact the manager's performance if left on in a normal network. These commands do not persist through reset or power cycle.

Repeat this several times and measure the roundtrip latency (delta between TX time and RX time, here 1382 ms)

- Turn on bidirectional backbone:

```
> set config bbsize 2

> set config bbmode 2
```

After this, a manager reset is required for the backbone to come on.

- After reconnecting to the manager and joining all motes, turn CLI latency trace back on:

```
> trace stats on
```

- Repeat the *sendData* test of step one - you should see a dramatic improvement in roundtrip latency.

The powered backbone persists through reset/power cycle until deactivated and the network is reset again. It results in the motes consuming 1-2 mA average current.

4.4 Channel Hopping and Range

Unlike most other 15.4 based systems, Dust's products employ a Time Slotted Channel Hopping MAC - this means that transmissions are spread out over all (or some - see Blacklisting below) channels in the 2.4 GHz band. This has the advantage that the system's performance depends on the average channel stability, rather than a single channel's stability. The following test will show you the how to measure single channel stability and see the effect of channel hopping. This test requires that you have access to a 802.11g Wi-Fi router and can set it on a particular channel - set the router to 802.11 channel 6 - this should cover mote channels 4–7. You will also need to configure your manager and a mote for radio test.

Note that in the various APIs below, channels are numbered 0-15. These correspond to channels 11-26 in the IEEE channel numbering scheme.

4.4.1 Using radiotest for Single Channel Measurements

This description places the manager in the role of transmitter, and the mote in the role of receiver, but the test can be run easily with roles swapped.

- On the manager CLI, log in and place the manager into radio test mode

```
> login user
> radiotest on
> reset system
System reset by CLI
SmartMesh IP Manager ver 1.1.0.30.
   658 : **** AP connected. Network started
```

- After the manager resets, configure it to be the transmitter, setting it for a packet test, on channel 0, at +8 dBm power, 200 packets, 80 byte payload, 5 ms interpacket delay - wait to hit return until you've configured the mote as receiver. Login is optional in radiotest mode.

```
> radiotest tx pk 0x0001 8 200 80 5000
```

- On the mote CLI, place into radio test mode and reset. Then configure it to receive on channel 0 for two minutes

```
> radiotest on
> reset
SmartMesh IP mote, version 1.1.0.41 (0x0)


> radiotest rx 0x0001 120
```

- At this point hit return on the manager to start the test. After the manager is done sending, record the number of packets received on the mote CLI:

```
> radiotest stat
Radio Test Statistics
OkCnt : 998
FailCnt : 2
```

The ratio of received to sent is the path stability (or packet success ratio) for this channel. Note that 1000 - *OkCnt* may not equal *FailCnt*, since the manager only counts packet that it can lock onto.

Repeat for all channels. You should see a dip in path stability around the channels being used by the Wi-Fi router - note how the dip isn't as big as you might expect. This is in part due to the fact that the router is also duty cycling, and is only on a small fraction of the time.

4.4.2 Range Testing

The radioTest commands can also be used to do range testing. Range is the distance at which two devices can reliably exchange packets, but it is also a grey area, since the following all affect the ability of two devices to communicate at a distance:

- Reflectors - things the radio signal bounces off of, including the ground.
- Obstacles - things the radio signal must go through
- Presence of interferers
- Antenna design
- Radio power settings

The simplest test is to repeat the single channel measurement described above, and repeat for all channels, since a real network will use all channels. You will find that the packet error rate is strongly influenced by the height of the radio off the ground, due to self-interference between the line-of-sight path and the bounce path. With motes at 1 m elevation, you may see a drop off in packet success ratio near 50 m spacing, only to have it improve farther out. Motes placed several meters above an open field should be able to communicate at hundreds of meters.

After this test, return the manager and mote to normal operation:

```
> radiotest off
OK
> reset
```

The DC9003 boards come with an integrated chip antenna that has significantly less gain (-2.3 dBi average vs +2 dBi) than a dipole antenna. Range measurements should take this difference into account.

4.4.3 Blacklisting

The manager provides an API to blacklist certain channels in the case of a known interferer. In general, unless you know that there is a strong interferer (such as a very busy wifi router) you should not blacklist

- Form a 5 mote network - after all motes have joined, clear statistics:

```
> exec clearstat
```

- Run for an hour. Use the manager CLI to look at the path stability for the network as a whole. At no time should you see packets lost due to the interferer.

```
> show stat

Manager Statistics --------------------------------

   established connections: 1

   dropped connections    : 0

   transmit OK            : 1907

   transmit error         : 0

   transmit repeat        : 0

   receive  OK            : 35

   receive  error         : 0

   acknowledge delay avrg : 15 msec

   acknowledge delay max  : 35 msec

Network Statistics --------------------------------

   reliability:  100% (Arrived/Lost:   1931/0)

   stability:     98% (Transmit/Fails: 983/25)

   latency:      200 msec


Motes Statistics ----------------------------------

   Mote    Received   Lost  Reliability Latency Hops

   #2          1019      0   100%         120    1.0

   #3            19      0   100%        1050    1.0
```

- Reset the manager, blacklist channels 4-7, and reset the manager. The blacklist persists through reset/power cycle.

```
> login user
> set config channellist 00007f0f
> reset system
System reset by CLI
SmartMesh IP Manager ver 1.1.0.30.
   658 : **** AP connected. Network started
```

- After the network has formed, clear stats, run for another hour, and observe the path stability - it should have improved.

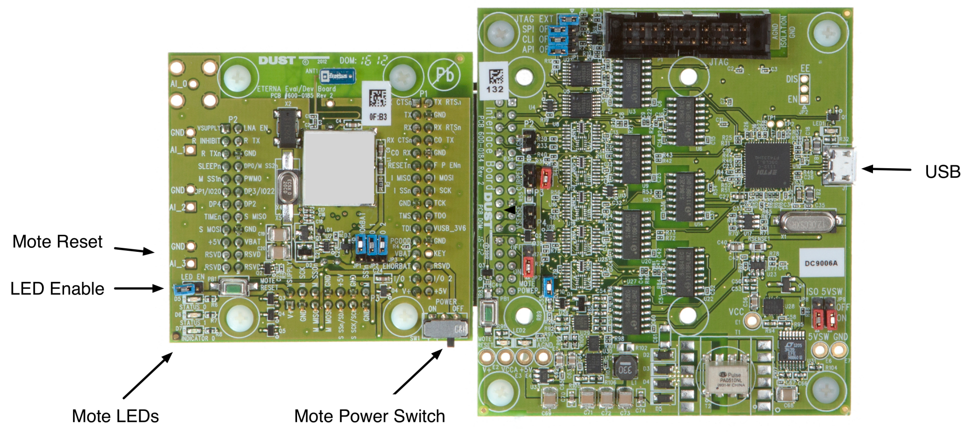
Return the manager to 15 channels before proceeding:

```
> login user
> set config channellist 00007fff
> reset system
System reset by CLI
SmartMesh IP Manager ver 1.1.0.30.
   658 : **** AP connected. Network started
```

4.5 Power

This section details how to measure power on an Eval/Dev board. In order to measure mote current, you should use a DC9003 A-B/DC9006 combination. You can either connect a scope across the jumper marked VSENSE on the DC9006 - this measures the voltage drop across a 10 Ω series resistor, or you can remove the red jumper (JP5) behind the CURRENT header, and connect an averaging current meter across the VSENSE pins.

⚠ The LED_EN jumper on the DC9003A-B board must be disabled or the current measurement will be incorrect.



Suggested Measurements:

⚠ If the mote is in **slave** mode, it will continually generate boot events until they are acknowledged, drawing > 500 µA of current. You must connect APIExplorer from the SmartMesh SDK, which acknowledges this event, to properly measure idle current.

- Idle current - when a mote has been reset, but no join API command has been issued. The mote must be configured in **slave** mode to prevent automatic joining. After measuring idle current, return the mote to **master** mode.
- Searching - Measure at 5% duty cycle, and 100% duty cycle. You should see the current change from ~250 µA average to 5 mA average.
- Mote joined - Set *dnfr_mult* to 4, and *setDnframe* to fast and reset the manager before joining the mote. After an hour, the manager will transition to the longer downstream frame, allowing you to compare the average current. It should be about 43 µA before the transition, and 30 µA after. Turn off advertising with a 'exec setAdv off' command in the CLI and see that the average current falls to 14 µA.

- Backbone - Set the bbmode to 2 and bbsize to 2 and reset the manager. Measure the average current - it should be about 500 µA.

Dust provides a SmartMesh Power and Performance Estimator to help estimate mote average current under various use cases. Compare your test results to the spreadsheet results.

4.6 Mesh Behavior

Dust's networks automatically form meshes - each mote has multiple neighbors through which it can send or forward data. This setting is a manager policy that is visible with the manager CLI command >show config (see above). The config element that sets this policy is numparents, and as you can see above the default value is 2. The manager will make sure every mote has at least the number of parents indicated by numparents, provided there are no loops in the mesh. Try to sketch this on your white board and you'll find that in a one mote network, that one mote will have only the AP mote as a parent. Add a second mote, and one mote will get two parents, but the other will be left with only one parent, to avoid making a loop. Regardless how many motes you add, there will always be one 'single parent mote'.

4.6.1 Testing the Mesh

The Stargazer GUI application will allow you to see the mesh as it forms - if you haven't already installed it, do so now. It also requires that you install the Serial Mux.

A mote having two parents is the best balance between robustness and power consumption under most operating conditions. Robustness is achieved by having multiple parents. To confirm that we recommend you find a mote in your network that is a parent to one or many motes. Power that mote off. Data delivery from that mote will stop, but the children will continue to send data. Use 'trace stats on' as above to show a live stream of data from all the children motes, which are sending data through their other parents. You may see a temporary increase in latency for a few minutes, and if you capture that trace to a file you can show this graphically. The manager will take a few minutes to reassign parents and links in the system to account for the loss of the mote you powered down.

4.6.2 Changing Mesh Policies

The parameter 'numparents' can be given any of 4 values: 1, 2, 3 or 4. If you set numparents to 1, your network will be a tree, not a mesh. Each mote will have one parent, and that one parent may not be the AP. This is better than a star in that a tree supports multi-hopping. But like a star, a tree network has numerous 'single points of failure'. If you power down the parent of a mote in a tree, the child mote will drop off the network because it has no connections to any other devices. It will reset and will have to join again, most likely resulting in data loss. A tree structure can be slightly lower power, because each device only has to listen to one parent for downstream messages and only has to maintain synchronization with one parent. Furthermore it is assumed that data latency will be more bounded in a tree, because individual packets can't 'wander' around the mesh. We believe that the likelihood of network collapse and data loss far outweigh the potential benefit in power and latency.

Two is the default and has a nice balance of robustness vs power. Setting numparents to 3 or even 4 increases the 'meshiness' of the system and gives the network additional robustness at the cost of somewhat higher power. We recommend increasing the number of parents in networks where you know that paths will be breaking frequently. See the Application note Identifying and Mitigating the Effects of Interference. Moderate mobility of some nodes may be better supported with numparents = 4.

4.7 Data Rates

The SmartMesh SDK contains two sample applications: TempMonitor, which allows you to set the rate of temperature publication on one or more motes in a SmartMesh IP network, and PkGen, which allows you to generate packets of a particular size and at a particular rate. Both of these applications can be used simultaneously to generate both temperature and simulated data, at different rates, at the same time. The Stargazer application can also be used to configure temperature reporting as well as analog and digital data.

With these applications you can answer the following questions:

- What does real data flow in a network look like?
- How fast can a single mote go?
- How fast can all motes go? With upstream backbone on?

Remember final data rates depend upon the number of motes, the services each mote is asking for (or base bandwidth in addition to or instead of services), topology, and path stability, but this tool gives a good sample of the kind of flexibility you can expect.

5 Application Note: How to use the Embedded Manager Subscribe API Filters

5.1 The Subscribe Filter

The *subscribe* API is used to configure which notifications the manager should send to a host application. The list of notificiations is specified in one of two bitmaps:

- The *filter* bitmap specifies notification to be sent.
- The *unackFilter* bitmap specifies notifications to be sent using unacknowledged (best-effort) communication. The manager will send each notification as it is generated, regardless of host behavior. The default behavior is to send using acknowledged communication, in which subsequent notification packets will be queued while waiting for the host to acknowledge each notification.

The bits in the *unackFilter* bitmap have no meaning unless the corresponding bit is set in the *filter* bitmap.

5.2 Acknowledged or Unacknowledged?

In cases where the link is unlikely to have errors, the host UART is always available and has sufficient buffers (e.g. when the host is a PC), or the application can tolerate some lost data (e.g. a missed report does not result in an alarm condition), then unacknowledged communication may be used for *data* and *ipData* notifications. Other notifications are far less frequent and more important not to miss, so they should always use acknowledged communications. Unacknowledged communication is faster than acknowledged, since the inter-packet delay can be as small as a bit-time (~10 µs), and no time is spent sending the acknowlegement. Given current serial port speeds (115200 bps), it may be necessary to use unacknowledged communications to maximize Manager data throughput, and you should consider using it when more than 10 packets/s are being generated.

The acknowledged transfer mechanism ensures that the host receives each packet in the face of certain errors:

- framing errors (typically caused when the host is asleep when the packet starts)
- bit errors (caused by a link with insufficient SNR, e.g. from a poorly shielded cable)
- host out of Rx buffers or isn't ready to receive for another reason

The manager waits 200 ms before retrying each packet. Once a packet has been retried 3 times without acknowledgement, the manager will consider the session dead, drop the pending packet (and any other queued notifications) and disconnect. The host will need to re-establish the session as described in the SmartMesh IP Manager API Guide and re-subscribe to notifications.

The manager will queue a small number of packets before refusing packets from the AP. Depending upon generation rate and path stability, the network may not be able to tolerate the maximum number of retries on each packet before the motes' queues will fill and they will begin refusing packets. This may result in lost or stale packets. If the aggregate packet rate is < 1 packet/s, the network will never drop packets due to notification retries.

6 Application Note: Monitoring SmartMesh IP Network Health with the Embedded Manager

Embedded Manager IP networks maintain a minimal set of statistics needed to manage and optimize the network. They do not aggregate statistics for tracking historical trends, and per-mote granularity is not often available. However, the source of much of this statistical information, namely raw mote health and state reports, is available in the form of notifications to which the client application can subscribe in order to provide network health and debugging feedback to the user. Some examples:

- By watching state changes, the application can tell if any motes are resetting.
- By watching neighbor health reports, the application can make sure that all motes have a sufficient number of good neighbors for recovery if paths fail.

Additionally, the application can use the manager API to get more information about the network. This document details several tests that can be continuously run on functioning networks to monitor their health.

This process is broken up into two parts. The first is the set of notifications that the application should be continuously monitoring upon their output from the manager. The second is a set of API calls that the application should make periodically to gather the rest of the required information. Ideally, the application is started at the same time as the network and monitors health for the entire lifetime of the network.

We recommend storing all notification data (perhaps breaking it into daily logs). If storage is a concern, then storing max, min, and FIR filtered average for each item (e.g. path x RSSI) may be sufficient for providing user feedback.

⚠ With LTC5800/590x Managers, Mote IDs are NOT guaranteed to be the same after a manager reset. Your application will need to match Mote ID to Mote MAC address in case of manager reset if you with to present consistent lifetime statistics.

6.1 Health Reports

There are three types of Health Report (HR) sent by each mote in the network, and each is sent once every 15 minutes. We only need to monitor the Neighbors Health Report and Device Health Report. The third type, Neighbors Discovery Health Report, provides information that becomes more easily accessible through the manager API. The application should monitor HR notifications as they come asynchronously from the manager.

6.1.1 Neighbors HR

This HR gives us information about all the used paths that the mote is involved in. From this HR, we want to capture: *neighborId*, *rssi*, *numTxPackets*, *numTxFailures*. Also note the time stamp on the notification.

In typical use, most paths will be more heavily used in the direction from child to parent, though both motes report statistics on the path. In order to have enough statistical significance in a path stability, we should consider only paths for which *numTxPackets* > 10. In this case, the stability in percent is 100*(1-*numTxFailures*/*numTxPackets*). We want to record this stability along with the *rssi* of this path. We record these two items as a pair for each path.

6.1.2 Device HR

This HR provides information about the internal operation of the mote. Here we want to pull out values that tell us how successful the mote was in receiving messages from the application. This will factor into our calculation of overall network availability later. From this HR, we want to capture: *numTxOk*, and *numTxFail*.

The availability of the mote, in percent, is 100*(1-*numTxFail*/(*numTxFail*+*numTxOk*)).

For calculating the overall network availability, define two new variables and initialize them to zero. Call them *appTxPk* and *appTxFail*.

To keep a running tally with each Device HR:

- *appTxPk += numTxOk + numTxFail*
- *appTxFail += numTxFail*

6.2 Periodic API Calls

We recommend polling the manager every 15 minutes with these API calls to match the period of the HR. Iterate through *getNextPathInfo* starting with 0 for each mote.

6.2.1 getMoteConfig

Store: *macAddress*, *moteId*, *isAP*

This command is a convenient way to learn the MAC address for every mote present in the network. Iterate through this to get all the MAC address/Mote ID pairs by using the *next* flag, and start with a *macAddress*=0 to kick things off. Storing this address correspondence allows the application to decipher the rest of the health information which can be reported either by MAC address or by Mote ID. Generally the information returned by this API will not change providing no new motes are added to the network.

6.2.2 getMoteInfo

Store: *numGoodNbrs*, *requestedBw*, *totalNeededBw*, *assignedBw*, *packetsReceived*, *packetsLost*

The information returned by this API is continuously updated by the manager as it receives packets from the mote. Once all the possible connectivity of the network has been fully discovered, the *numGoodNbrs* value will remain static, but optimization can change the bandwidth distribution requirements throughout the network.

For calculating the overall network reliability, define two new variables and initialize them to zero. Call them *networkRX and networkLost*.

To keep a running tally with each *getMoteInfo* call:

- *networkRX* += *packetsReceived*
- *networkLost += packetsLost*

6.2.3 getNextPathInfo

Store: *source*, *dest*, *direction*, *numLinks*, *quality*, *rssiSrcDest*, *rssiDestSrc*

The manager maintains all of the path information, but only in a coarse and averaged form. You can get all of this information at a more precise resolution by watching the HR activity.

6.3 Tests

These are the tests that can be run on all the collected data to provide warnings about network health.

The tests below list conditions that should raise warning flags, i.e. you want your network to **not** have any of these conditions.

To calculate the number of upstream links, look at the *direction* reported in each path.

- total TX links sums up the *numLinks* of paths with *direction*=2
- total RX links sums up the *numLinks* of paths with *direction*=3
- total number of parents is the count of paths with *direction*=2

6.3.1 For the AP Only

These tests run only on the AP. This device is identified by having the *isAP* flag set.

AP Close to Link Saturation

Sum up the *numLinks* on all *direction*=3 paths as described above. An AP without external RAM can support 150 RX links, so anything above 140 RX links indicates that any additional services are in danger of not being granted. An AP with external RAM can support 250 RX links, so 230 RX links is a good threshold to raise a warning.

6.3.2 Iterate Over All Motes

These tests are run on each mote individually.

Fewer Than 3 Good Neighbors

Every mote in the network should have three "good" neighbors, *i.e.* having a quality score over 50%. The manager keeps track of the count of good neighbors, so verify that the manager reports at least 3 good neighbors for each mote in the *numGoodNbrs* field.

Mote Close to Link Saturation

Sum up the *numLinks* on all paths as described above to add together the number of upstream TX and RX links. Since Eterna motes can store 200 links total, A mote having more than 150 links can indicate a danger of a bottleneck developing.

More Joins than the AP

Count the number of times that a mote has transitioned to the **Oper** state, and count the same number for the AP. If the mote has joined more times than the AP, it means the mote has reset and rejoined. Mote resets, especially if they happen in groups, are the biggest indication that there is something wrong with the network, such as interference, motes being too far apart, or a hardware issue that causes resets.

More than One Single-Parent Mote

There should be exactly one single-parent mote, and this mote should have the AP as its parent. If not, there is insufficient connectivity for some motes in the network. Consider adding repeater motes near the parents of any other single-parent motes.

Insufficient Bandwidth

The manager keeps track of the requested and assigned bandwidth for each mote. The bandwidth reported here is the average time between links on the mote, so a lower number means more links per second. Check to make sure that *totalNeededBw* > *assignedBW.*

Other tests

- Motes also report details about their queue occupancy. This additional data can be used to run more health checks.
- The *getMoteInfo* API returns information about average latency. Monitor this over time and flag any case where it rises unexpectedly.

6.3.3 Iterate Over All Paths

The HR notifications capture 15-minute snapshots of every path in the network. Iterating over these snapshots can help to identify bad paths.

Bad Stability-RSSI

Flag any path that has either:

- RSSI above -80 dBm and stability < 50%
- RSSI above -70 dBm and stability < 70%

These paths lie beneath our prototype RSSI-Stability waterfall curve. If there are occasional outliers here, it is not a large concern. Consistent points below either threshold for a mote can indicate a hardware problem or interference in the vicinity. See "Application Note: Identifying and Mitigating the Effects of Interference" for more details.

As we captured the time of the notification from each HR, it is easy to print out the times of the outlier points to see if they are temporally grouped.

6.3.4 Network Checks

These are properties that can be checked for the network as a whole.

Reliability < 99.9%

The manager maintains a coarse measure of overall reliability, rounded to the nearest percent, in the *netReliability* returned from the *getNetworkInfo* API. Often reliability is expressed as a "number of nines" metric, so precise values are required. SmartMesh networks are designed for three nines of reliability or more.

The total reliability, calculated from our variables above, is 100.0*(1-*networkLost*/(*networkLost* + *networkRX*)).

Availability < 99%

To get the average availability for the network, take the mean of the per-mote availability measured earlier.

6.4 Graphing

If the application is continuously running, each of the quantities above could be plotted with 15-minute resolution. We have previously seen daily rhythms in network stability and latency in networks deployed in industrial environments where the amount of interference and moving machinery varies greatly between the workday and nighttime. Often tracking these metrics over time is much more revealing than taking a single snapshot.

7 Application Note: Data Publishing for SmartMesh IP

This note describes the specific steps required for an OEM microprocessor's firmware application to connect to a mote in the network and then have it send data over the wireless network. The key concepts from the SmartMesh IP User's Guide and SmartMesh IP Mote API Guide are brought together here. The User's Guide describes the details of how the mote and the OEM microprocessor interact and combine to form a system.

7.1 Request and Response Packet Formatting

All packets must use HDLC Encapsulation as described in the SmartMesh IP Mote API Guide. This means that the packet is preceded and followed by a framing byte 0x7E. A 2-byte frame check sequence (FCS) bytes must be computed for the packet and appended before the trailing framing byte. Instances of 0x7D and 0x7E in the payload or FCS require escaping, as described in the Mote API Guide in the "Protocol" section under "Packet Format."

Flag	Payload	FCS	Flag
7E	Packet Payload	2 bytes	7E

7.1.1 Header

The payload of both response and request packets begins with a 3-byte header. The first byte indicates the type of command or notification being sent or responded to. The next byte is the length of the remaining payload (not counting header or response code). The third byte is a set of flags. A complete description of the header and flags can be found in the Protocol section of the mote API guide under "Packet Format."

Request

Header	Payload
3 bytes	Request Payload

Response

Header	Response Code	Payload
3 bytes	1 byte Response	Response Payload

The length byte does not include the 3-byte header, or the response code.

7.1.2 Flag Byte

The third byte in the three byte header is the flag byte. The flag byte currently has only 3 usable bits:

Bit 0 - Request/Response: Bit 0 is cleared (0) if the packet is a request and is set (1) if the packet is a response. Acknowledgement (ACK) packets are response packets and must have this bit set.

Bit 1 - Packet ID: For every new request packet by the OEM micro, the packet ID bit must be toggled.

Bit 3 - SYNC bit setting rule: The SYNC bit must be set (1) on the first request packet OEM micro sends to the mote. It must be cleared (0) for subsequent requests. The SYNC bit is set (1) on boot event packet, which is the first request packet sent from the mote to the OEM micro.

7.2 Basic Steps

There are a total of 7 basic steps that the OEM micro needs to perform in order to get the mote to join a network and start publishing data. The setup of the OEM micro's serial port is a very important first step in order for the rest of the communication to be successful. To generate sample firmware for a simple publish application, we need only to code these in order to get the combined micro+mote system to start publishing data. The steps that follow are:

1. Set up the serial interface
2. ACK the mote boot event
3. Perform pre-join mote configuration
4. Issue *join* command
5. Monitor join progress
6. Publish data
7. If a *boot* event is received, go to 3.

1. Set up the Serial Interface: The communication between the OEM micro and the mote takes place on the API serial port using a 4-wire protocol (UART Mode 4: TX, RX, UART_TX_CTSn, UART_TX_RTSn lines) by default. By default, the API port settings are: baud rate is 115.2 Kbps, 8 data bits, no parity, 1 stop bit.

The baud rate on the LTC5800-IPM can be changed to 9600 if required, by updating the fuse table programming on the mote.

2. ACK the Mote Boot Event: Whenever the mote is powered up (or reset) it sends a boot event packet on the API port. The boot event packet is as follows:

```
7E 0F 09 08 00 00 00 01 01 00 00 00 00 D7 67 7E
```

The mote will continue to send this packet until it is explicitly acknowledged by the OEM micro. If the serial port settings are correct then the micro should be able to see this packet show up in its receive buffer. If the micro's serial port is configured incorrectly, one can see this packet on a scope or Logic Analyzer on the Tx pin of the mote. It usually takes about a second after power up to receive this packet.

Note that different versions of mote software may start with the packet ID bit set to either 0 or 1 - this means the flag byte could either be 0x08, or 0x0A on the boot event.

The ACK packet for this event is:

```
7E 0F 00 01 00 FF 57 7E
```

The packet ID of the ACK should match that of the incoming packet, so could be 0x01 or 0x03 here (for a packet with flag bytes 0x08 and 0x0A, respectively).

The OEM micro needs to respond with this packet in order for the mote to stop sending the boot event packet. Responding with this ACK will move the mote from state 0 (**Init**) to state 1 (**Idle**).

3. Perform Pre-Join Mote Configuration: Once the mote boot event has been acknowledged and it is in the **Idle** state, it is ready to join the network. Before issuing the *join* command, you may want to change one or more of the pre-join configuration settings. These settings may be left as the factory default at first, but should be later adjusted for optimal operation. Some of these configurable parameters are:

- *setParameter <networkId>* - The default Network ID is 0x04CD (1229).
- *setParameter <joinKey>*: The default join key is 0x 445553544E4554574F524B53524F434B. For the highest level of security, each mote should have a unique join key.
- *setParameter <joinDutyCycle>*: Default is 0xFF or 100% (of 255)

See the Mote API Guide - the "Commands" section details each API, and the "Definitions" section gives the encoding for arguments.

Assuming it had been set previously to a different value, the packet to change the *joinDutyCycle* parameter to 0xFF or 100% (255) is:

```
7E 01 02 08 06 FF 2F 60 7E
```

Header = 01 02 08, so this is the *setParameter* command (0x01), length of payload = 2 bytes (0x02), and the flags byte has bit 3 set (SYNC, request, packet ID = 0)

Payload = 06 FF, so this is the *joinDutyCycle* being changed (0x06), and the value is 100%, i.e. 255 (0xFF)

Here we have this as the first command sent to the mote, so the SYNC bit is high. If this is not the 1st request packet the OEM micro is sending to the mote, make sure to unset (bit 3 = 0) the SYNC flag.

The mote will then reply with this response:

```
7E 01 01 09 00 06 A0 21 7E
```

Header = 01 01 09, so this is the *setParameter* command (0x01), length of payload = 1 bytes (0x01), and the flags byte has bits 3 and 0 set (SYNC, response, packet ID = 0)

Response code = 00, so the command had no errors (0x00)

Payload = 06 , so this is the *joinDutyCycle* being changed (0x06)

4. Issue *join* Command: Now that the pre-join configuration is done, the OEM micro is ready to issue a *join* command to the mote that will prompt it to enter the process of joining a network (specified by its Network ID). Assuming that this is not the first packet to the mote (Here SYNC=0), the join packet is:

```
7E 06 00 02 07 33 7E
```

The mote will respond with an acknowledgement:

```
7E 06 00 03 00 2C 9D 7E
```

5. Monitor Join Process: In the process of joining the mote will send several event notifications to the OEM micro that must be acknowledged. These notifications are as follows:

mote *joinStarted* notification:

```
7E 0F 09 02 00 00 01 00 03 00 00 00 00 C6 DB 7E
```

OEM micro acknowledgement:_

```
7E 0F 00 03 00 4F 64 7E
```

mote *operational* notification:

```
7E 0F 09 00 00 00 00 20 05 00 00 00 00 A5 A2 7E
```

OEM micro acknowledgement:

```
7E 0F 00 01 00 FF 57 7E
```

mote *svcChange* notification:

```
7E 0F 09 02 00 00 00 80 05 00 00 00 00 29 7A 7E
```

OEM micro acknowledgement:

```
7E 0F 00 03 00 4F 64 7E
```

6. Publish Data: To send your own packets, you will need to use the *sendTo* command, as described in the SmartMesh IP Mote API Guide. The mote uses a network socket interface to send data to the manager or an IP host on the internet. Before you can send data to a particular destination, you must open and bind a socket to the destination. The default destination for packets should be the Manager. This process is described in the SmartMesh IP User's Guide in the "Communications" section.

These steps are:

1. Call *openSocket* command to open a communication socket - this will give you a *socketID* for the socket. Currently only UDP sockets are supported.
2. Call *bindSocket* command to bind the socket to a port, the *destPort* you will use in the *sendTo* command.
3. Use *sendTo* command to send data. You will need to specify a *serviceType, priority, and packetId*, in addition to the payload and socket information. These are discussed in the *sendTo* documentation in the Mote API Guide. Currently only bandwidth (as opposed to latency) services are supported. Repeated calls to *sendTo* can be made on the open socket.
4. Call *closeSocket* command when you will no longer need to send data to that destination. This removes the port binding and frees any memory associated with the socket. It is not required to close a socket after each packet.

Each step will require that the micro send a packet to the mote and then receiving the acknowledgement.

1. **Call *openSocket*** - This will open a UDP socket:_

```
7E 15 01 00 00 F4 0B 7E
```

Mote acknowledgement, returning socket ID 22 (0x16):

```
7E 15 01 01 00 16 B3 6E 7E
```

2. **Call *bindSocket*** - While any port can be used, payload is maximized when a port in the range of 0xF0B8-0xF0BF is used. Here we bind the previously obtained UDP socket (22) to port 0xF0B8.The mote and the destination need to agree on which port is being used - unless you have a specific reason to use another port, 0xF0B8 is a good default choice.

```
7E 17 03 02 16 F0 B8 D3 9B 7E
```

Mote Acknowledgement:

```
7E 17 00 03 00 26 42 7E
```

3. **Call *sendTo*** - In this packet, sample data 0xAABBCCDDEE is the *payload* to be sent to the manager (*destAddr* = 0xFF020000000000000000000000000002) with packet ID 0 (0x0000), See the SmartMesh IP Mote API Guide for details on other fields. Note that for a different payload the FCS would be different._

```
7E 18 1C 00 16 FF 02 00 00 00 00 00 00 00 00 00 00 00 00 00 02 F0 B8 00 00 00 00 AA BB CC DD EE AF
B2 7E
```

Mote Acknowledgement:

```
7E 18 00 01 00 7F C3 7E
```

After you send the packet to the Tx queue, the mote will send a notification indicating when it has finished the transmit. This will need to be responded to as well.

Mote *txDone* notification with packet ID 0 (0x0000):

```
7E 25 03 00 00 00 00 A4 7B 7E
```

Micro acknowledgement:

```
7E 25 00 01 00 02 04 7E
```

If you set the packet ID in the *sendTo* command to 0xFFFF, a *txDone* notification packet will not be generated.

4. **Call *closeSocket*** - When you are done sending messages to this destination, it is recommended that you close the socket you were using. This is more common when the communications is in response from an message from an internet host, where a finite "transaction" is taking place. For a sensor configured for periodic publishing, the socket would only be closed should the micro need to reset.

This can be done with the *closeSocket* command as follows:

```
7E 16 01 02 16 3E 68 7E
```

Mote acknowledgement:

```
7E 16 00 03 00 8D 5E 7E
```

7.3 Next Steps

Now that you have successfully sent a data payload to the manager you can look at the mote API guide for more commands and detailed descriptions of customization options.

- Bandwidth Services - Once *operational*, the mote is available to send the data through the wireless network. By default, an IP network is configured such that every mote can publish data at 9 second intervals (the *base bandwidth* for an IP network) without requesting a service. If this will always be sufficient, the OEM micro need not request any services from the manager. Should the application require sensor data (like temperature, humidity, voltage, current) faster than every 9 s where all motes publish at the same rate, it is a *homogeneous bandwidth* network, and base bandwidth may be increased. If the application calls for publishing data at different rates for different devices, this is a *heterogeneous bandwidth* network and all devices should request services. We recommend that OEM integrators always use services, since they are simple to implement and offer the most flexibility. Services are discussed the "Services" section of the SmartMesh IP User's Guide, and "Bandwidth and Latency" section of the SmartMesh IP User's Guide.

- Test Radio Commands - these can be used for manufacturing tests to verify the top level assembly, e.g. that the antenna has been properly connected. The test radio commands (*testRadioTxExt* and *testRadioRx*) are described in detail in the .SmartMesh IP Mote API Guide.

- Sockets and UDP ports - these are discussed further the "Communication" section of the SmartMesh IP User's Guide.

8 Application Note: Managing Advertising in an Embedded Manager Network

Unlike the WirelessHART managers, the SmartMesh IP Embedded Managers do not control advertising in response to network conditions. Advertising costs each mote ~14 µA average current, which can be significant to low-traffic motes. The LTC5800-IPR presents a *setAdvertising* API for enabling/disabling advertising, so on cases where this is overhead is unacceptable, the host application can implement the logic to control advertising. This application note presents an example state machine for this purpose.

When advertising is off, new motes, or motes that have gone **Lost**, cannot find and join the network. For that reason we recommend leaving advertising on.

8.1 Advertising State Machine

Advertising comes on automatically when the manager starts.

The state machine should have the following logic:

- Leave advertising on when no motes are present - until there is a *moteOperational* event for a device other than the AP, advertising should be left on. Waiting for the *moteOperational* notification (rather than *moteJoin*) ensures that a mote that starts but fails to complete joining will still be able to hear the network.
- Deactivate after the "last" mote has joined - there should be a suitable timeout after the last *moteOperational* event, e.g. 1 hour.
- Reactivate when a mote is lost - advertising should be restarted when either a *moteLost* or *moteReset* event is generated.
- Deactivate when no lost/reset mote joins within a timeout - for lowest power operation, there should be a suitable timeout after the last *moteLost* or *moteReset* event, e.g. 1 hour. Note that a lost device that doesn't rejoin within this timeout will not be able to rejoin until advertising is re-activated for another reason.
- Reactivate advertising when a user wants to add more motes to the system.

9 Application Note: Improving Latency with the Embedded Manager Powered Backbone

9.1 Introduction

All SmartMesh managers have a variety of configurations that can be used to adjust key network performance metrics such as average mote power and message latency. There may be several different ways to tune a network to solve a particular problem – this document presents the Powered Backbone feature in the SmartMesh IP Embedded Manager and details when it should and should not be used in networks. Typically just called the backbone for short, this slotframe enables low-latency communication either upstream or in both directions. The upstream communication comes without additional energy costs at unpowered devices while enabling a bi-directional backbone requires increased power at all devices.

Limitations:

- There is no way to have a downstream-only backbone.
- Backbone mode and size are chosen before booting up the manager and cannot be changed without a manager reset.
- Power source settings presented at mote join determine backbone behavior - changing power source settings after join will not affect the backbone behavior once assigned.

Example Power calculations in this document are done with values from the SmartMesh Power and Performance Estimator.

9.1.1 General Motivation

The backbone was developed to achieve ZigBee-like low-latency upstream. In ZigBee networks, the routers are powered and all devices have to be within range of a router. In a SmartMesh IP network with the upstream backbone active (set using *bbmode*= 1), any mote can transmit in any slot provided it has a parent that is a powered node.

The powered nodes, if they themselves are not transmitting in a slot, will listen for packets from their children. As a consequence, unpowered devices will have a lot of idle TX events which do not cost any energy when using Eterna motes. Powered devices have a lot of idle RX events, and these do use energy because the radio has to activate in receive mode to be ready if a packet does come in. The upstream backbone could be set on a longer slotframe length, but we will see below that most applications work best when the backbone has a minimum-length 1-slot slotframe. The length is set through the *bbsize* parameter and the allowed values are 1, 2, 4, and 8 slots.

The downstream backbone works in a similar way, except now **all** devices need to listen and use energy. When the bi-directional backbone is active (set using *bbmode* = 2), even slots are used for the upstream backbone and odd slots are used for the downstream backbone. This even/odd separation ensures that traffic does not collide between the two directions which is critical for call-response traffic in imperfect RF environments. Dust command traffic does not use the downstream backbone, it is reserved for user traffic. The **only** bi-directional backbone length supported is 2 slots long.

All backbone activity happens on shared links. In parallel, the rest of the network continues to operate on higher-priority dedicated links. The backbone links are on a different channel offset than all other activity so even though traffic on the backbone might collide with other backbone traffic, it does not impact the reliable communication that still occurs on the dedicated links. Under no circumstances will the backbone decrease the overall reliability or increase the latency of a network.

9.1.2 Settings to Enable RX in the Upstream Backbone

SmartMesh IP motes report their maximum steady state current in their join request packet in the *maxStCur* field in the *powerSrcInfo* structure. The manager uses the highest setting, 0xFFFF, as the flag for enabling a mote to have RX links in the upstream backbone. This highest setting must be provided by the sensor application and should only be used if the mote truly is powered or has a very large battery supply, since motes with receive links in the backbone can draw > 1 mA. For simplicity, we will refer to motes with *maxStCur* = 0xFFFF as *powered* motes. All motes are permitted to have TX links in the upstream backbone, but only powered motes will be given RX links. A node is considered "powered" if the *maxStCurrent* field in the *setParameter<powerSrcInfo>* API is set to 0xFFFF, any other setting is unpowered. By default, nodes ship with maxStCurrent = 0xFFFE, which allows the manager to add links as needed without invoking the backbone - such a device will likely see < 500 µA average current in the busiest of networks. The distinguishing behavior of a powered node is that it gets receive links in the upstream backbone slotframe, there is no other difference between the schedules of powered and unpowered nodes.

9.2 Application 1: Low-latency Alarms

Network

- Backbone mode *bbmode*: 1
- Backbone slotframe size *bbsize*: 1

Expected Average Performance Metrics

- Latency: about 15 ms per hop
- Additional power at leaf nodes: none
- Additional power at routing devices: 925 µA (1-slot backbone)
- Total traffic limit: about 45 pkt/s

For a low-latency alarm network, we need to make sure that each unpowered mote is in range of a powered mote. The powered motes then form the **powered** backbone and any mote is at most one hop away from this backbone. As such, every mote can transmit in any slot. If the transmission fails, the transmitting mote assumes that the failure was due to a packet collision since the backbone slots are shared. The mote has a random exponentially increasing backoff to correctly use this contention-based set of links. There is nothing preventing the backbone from extending multiple hops from the AP. Taking average stability and the backoff mechanism into account, the mean latency per hop will be about 15 ms, or two slots. However, because the backbone can have collisions, the application must be able to tolerate the latency delivered by the parallel dedicated links in the worst case.

For routing purposes, the dedicated upstream and backbone links are treated equally. If a mote has an upstream packet and a dedicated upstream link to its parent in the next slot, it will transmit the packet on the dedicated link rather than the shared backbone link, and the parent will also listen on the dedicated link. This will necessarily be on a different channel offset so another child of the same parent trying to transmit on the backbone link will not collide with the first transmission. Similarly, if a mote has any other link, such as a join listen link or a downstream RX link, it will service this instead of servicing the upstream backbone link. Also, a packet going multiple hops could end up traveling some hops on dedicated links and some on the backbone links. All of this will happen as a random result of which link is available first for the mote and if any backoff occurs.

A mote will only have one backbone parent at a time, this will be one of the (usually two) upstream parents. The backbone links are not used to determine path failure, but if a path failure is detected on the dedicated links to the backbone parent, the mote will automatically start using the backbone to the second parent. The dedicated links were being used all along to this second parent, but the application again needs to be tolerant of longer-latency periods during path failures.

The upstream backbone is most enabling for a network that does not send much data but needs low latency when it does generate packets. For a 30-mote network with an average latency requirement below 100 ms, there just aren't enough slots at the AP to provide this with only dedicated links. If power is a concern, a longer *bbsize* setting of 2, 4, or 8 slots can be used. These reduce the routing device power, and increase latency, correspondingly.

9.3 Application 2: Call and Response

Network

- Backbone mode *bbmode*: 2
- Backbone slotframe size *bbsize*: 2

Expected Average Performance Metrics

- Round-Trip Latency: about 60 ms per hop
- Additional power at leaf nodes: 462 µA
- Additional power at routing devices: 925 µA
- Total traffic limit: about 16 pkt/s if the majority of motes are 1-hop

When sending packets to a single destination, the default SmartMesh IP network restricts packet ingress (downstream) to one per slotframe. With imperfect stability, this is less than one packet per two seconds. For applications with faster downstream requirements, the bi-directional backbone can be used. Contrary to the upstream backbone, the downstream backbone is not routing-equivalent. Internal command traffic does not use the downstream backbone links, and when the downstream backbone is active, user traffic is restricted to only use the downstream backbone.

This bi-directional backbone consumes at least 462 µA at all devices, and an additional 462 µA if the device happens to be an upstream router. As a rule of thumb, a packet should be able to go from the AP to a mote, and the response should come back in about 60 ms per hop. Because the backbone uses shared bandwidth, we recommend waiting for a response before sending the next call downstream. This 60 ms per hop is an average time and applications need to be able to tolerate outliers of up to 2 s.

9.4 Application 3: Lower Latency in All Low-Traffic Networks

Network

- Backbone mode *bbmode*: 1
- Backbone slotframe size *bbsize*: 1

Expected Average Performance Metrics

- Latency from one-hop motes: about 15 ms
- Additional power at motes: none

If the AP is not a power-constrained device, the upstream backbone can be used to decrease overall network latency without any increase in power at the motes. This is done by activating the upstream backbone and marking the AP as the only powered device. Consequently, only the one-hop children motes of the AP will have the backbone TX links, and remember that these do not cost the mote any energy when unused. Any one-hop mote that either generates a packet or receives one from a child is able to forward the packet on to the AP in the following slot. The latency in multi-hop networks will remain the same up to the first hop of the network, but since all packets must pass through the one-hop ring, all motes should see a decrease in their average latency with the upstream backbone active.

If there is significant traffic in the network (> 15 pkt/s total), there will be contention for the backbone links and transmissions will collide. The increased number of retries resulting from these collisions does impact the power at the 1-hop motes, though it does not decrease the throughput or increase the latency. For power-sensitive applications, we recommend using this 1-hop backbone only if traffic is known to be low.

9.5 Unsuitable Use of Backbone 1: Replacing Dedicated Links

Network

- Backbone mode *bbmode*: 1
- Backbone slotframe size *bbsize*: 8

Expected Average Performance Metrics

- Additional power at leaf nodes: none
- Additional power at routing devices: 116 µA

Having read all of the above applications, it may be tempting to use the backbone for everything. However, there are certain cases where the backbone is higher power and higher latency than the equivalent set of dedicated links. For example, suppose the network doesn't have any truly powered motes but we want to reduce the upstream latency a little from what we have observed in networks using only dedicated links. So we decide to report that all motes are in fact "powered" and increase the size of the upstream backbone to 8 slots with the idea that we can tolerate the extra 116 µA that this is going to cost at any mote with a child.

The inefficient part about the upstream backbone in this case is that the motes with children are having to listen as much as the AP to the backbone links, exactly once per 8 slots. When we assign cascaded dedicated links, no mote will end up with as many links as the AP. Since it is the idle RX links and not the idle TX links, an efficient network puts as much of the burden as possible on the AP. For the 116 µA that it costs, the same network could be built with a faster base bandwidth that would result in lower overall latency. And better yet, the trick discussed in Application 3 above can be used to further reduce the latency in this network for free.

In fact, if you are tempted to use an upstream backbone that is not the minimum length, you should carefully consider your options.

9.6 Unsuitable Use of Backbone 2: High-Traffic Networks

Network

- Backbone mode *bbmode*: 1
- Backbone slotframe size *bbsize*: 1
- 30 pkt/s total traffic to the Access Point
- 100 motes

Expected Average Performance Metrics

- Additional power at nodes: up to 120 µA

Remember that dedicated links take priority over the backbone links. If the network is busy, the Access Point will have dedicated links in most slots to receive the upstream traffic. Because the motes do not know the Access Point's full schedule, they will often transmit and fail on a backbone link which the Access Point is busy listening on a dedicated link. If there are several motes contending for the backbone bandwidth in this way, then even with backoff, most backbone transmissions will fail. If the mote is sending full-sized packets, all of these failures could result in spending up to 120 µA in extra transmit attempts. This may not be a huge penalty for a routing device, but it can be serious for a low-power device that is expected to use 30 µA total. This scenario may arise as a network grows - a network that starts out with a small number of motes reporting infrequently would see a huge latency improvement with the backbone, but as additional faster-reporting motes are added, the latency benefit to these original motes will decrease, and the backbone links that were originally free could now come at significant cost due as they repeatedly fail. Also note that the network would need to be reset in order to deactivate the backbone in such a scenario.

As stated in Application 3, the 15 pkt/s threshold is a good rule-of-thumb for deciding when the backbone will be beneficial or problematic. Again, having the backbone active will not increase the packet latency or decrease reliability in this busy network, it will just have a power penalty for the affected motes.

Backbone failures count as packet failures, so path stability values reported by motes in an overused backbone network will be very low. Watch for stability values below 50% to help diagnose this issue. These affected paths will be most apparent when drawing the RSSI-stability waterfall curve.

10 Application Note: Building Deep IP Networks

10.1 Introduction

With default settings, a SmartMesh IP network can span ~8 hops due to timeouts on source-routed packets. With a few changes (and sufficient connectivity) however, the SmartMesh IP family is well suited to building networks spanning a long linear distance - up to 32 hops (approximately 10 Km with line-of-sight placement). This includes applications such as monitoring the environment in a mineshaft and transmission line monitoring, where sensors tend to be deployed in the same direction away from an egress point. With a one-dimensional deployment in mind, packets from wireless sensor nodes ("motes") further from the manager will require more hops to get to their destination. We refer to these motes as being "deep" in the multihop network. There are some performance characteristics that are specific to such deep multihop networks, as compared to denser mesh networks:

1. The network will take longer to fully join.
2. The manager will be able to support lower total traffic limits. A maximum total egress of 10.5 packets per second should be respected.
3. More attention needs to be placed on deployment locations and connectivity.
4. Packet latency will increase with network depth.

10.2 Deployment Guidelines

As with all SmartMesh deployments, motes should all be deployed within range of three other potential parents to ensure network reliability. In the case of a linear deployment, this means all sensor motes must have three motes within range and closer to the manager. Furthermore, to preserve this property near the manager, we recommend placing some repeater motes (motes with or without sensors) close to the manager. If the radio range in the desired environment is *R*, then the deployment should be carried out according to Figure 1.

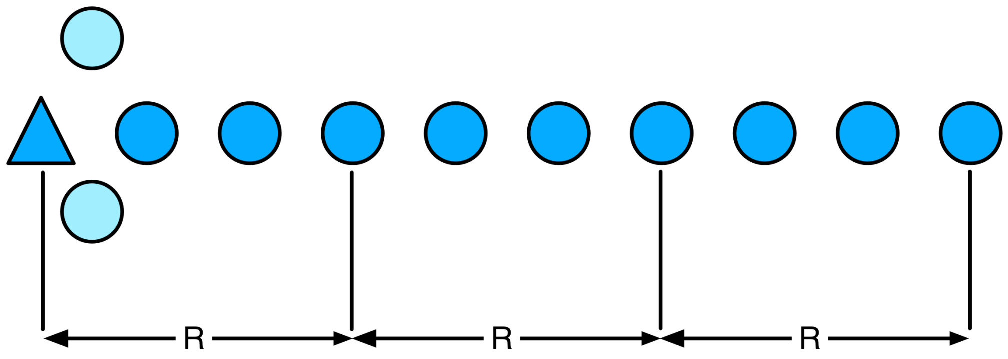


Figure 1. Each sensor mote (dark blue circle) has three upstream neighbors closer to the manager (triangle). The repeaters (light blue circles) provide traffic handling and spatial diversity.

If there are fewer sense points than this in the deployment, repeaters should be added to make up the required density of three potential parents per device. The total distance that can be covered is greatly affected by the deployment environment which affects range. As drawn in Figure 1, if for example *R*=100m, a 100-mote network can monitor out to beyond 3km. Placing the devices 20 to 30 meters off the ground and in line-of-sight can extend this range by 5x (i.e. > 15 km), and placing them in a mine shaft 1 meter off the ground can reduce it by half (1.5 km).

10.3 Determining Range

The two example deployments, mineshafts and transmission lines, are expected to lie at opposite ends of the device range spectrum due to the radio propagation characteristics in these very different environments. In either of these settings, there is no substitute for directly measuring device-to-device range with a pair of fully integrated motes using the actual finished or prototyped wireless sensor with the antenna you plan to use. This measured range informs both the number of repeaters and the maximum distance for the network. For this reason, we encourage OEMs to make range measurements as early as possible in the development cycle.

For more information on estimating range, see the application note "Planning a Deployment."

10.4 Mote and Manager Versions and Settings

Building deep networks requires SmartMesh IP Mote version 1.3 or later. If upgrading is necessary, it must be done before deployment. Additionally, the manager needs some configuration changes that are different from traditional deployments. These settings must also be changed before building the network and are persistent.

CLI settings for the VManager:

```
su becareful
set config network numParents=3
set ini RLBL_BCAST_TO=240000
set ini RLBL_FLOOD_TO=240000
set ini RLBL_MAX_TO=240000
set ini MINLINKS=8
set ini NUMMCAST=0
set ini RLBL_JOIN_TO_M=27
set config network topologyType=EVENT
set config network usFrameSize=256
set config network dsFrameSize=512
```

CLI settings for the Embedded Manager (requires version 1.2.1 or later):

```
su becareful
set config numparents 3
seti ini rlblbcto_f 240
seti ini rlblskto_f 240
seti ini rlblmaxto_f 240
seti ini rlblskto_s 240
seti ini minlinks 8 (refer to the calculation of L in the "Calculating Links" section)
seti ini iscascading 0
seti ini nummlinks 0
```

No configuration changes are required at the motes.

10.5 Calculating Links

At a minimum, the number of links assigned should be set to 8 (as in the settings detailed above). More links may be required for faster reporting rates or lower latency in the network, and can be calculated using the following formula:

L = [1.8M/T]

Where the number of links is *L,* the number of motes including repeaters in the network is *M*, and the reporting interval is *T.*

The square bracket here indicates that you should round up. We recommend limiting the total network egress to 10.5 pkt/s, so to get the maximum throughput for 100 motes, this is one packet per 10s per mote and a little extra to carry the health report packets. Calculating the link requirement in this case yields *L*=18.

Because of memory limits, restrict the maximum value of the product *LM* to 1800. For example, with our 100-mote deep network, the *L*=18 value is the maximum permitted. For a smaller network of 50 motes, we could set *L*=36 if we wanted to minimize packet latency. As in all SmartMesh networks, the user should keep in mind the tradeoff between adding more links and increasing the average energy use.

10.6 Estimating Latency

The latency distribution for each mote in a network is difficult to predict exactly. Consider for example the by-mote latency shown in Figure 2 from a 100-mote test with the Embedded Manager. While latency generally increases with mote depth, there is significant variation in the measured latency values, particularly in the maximum measurements. As an example from the plot, for the deepest mote the median latency is 10.2s and the 99th percentile latency is 20.6s.

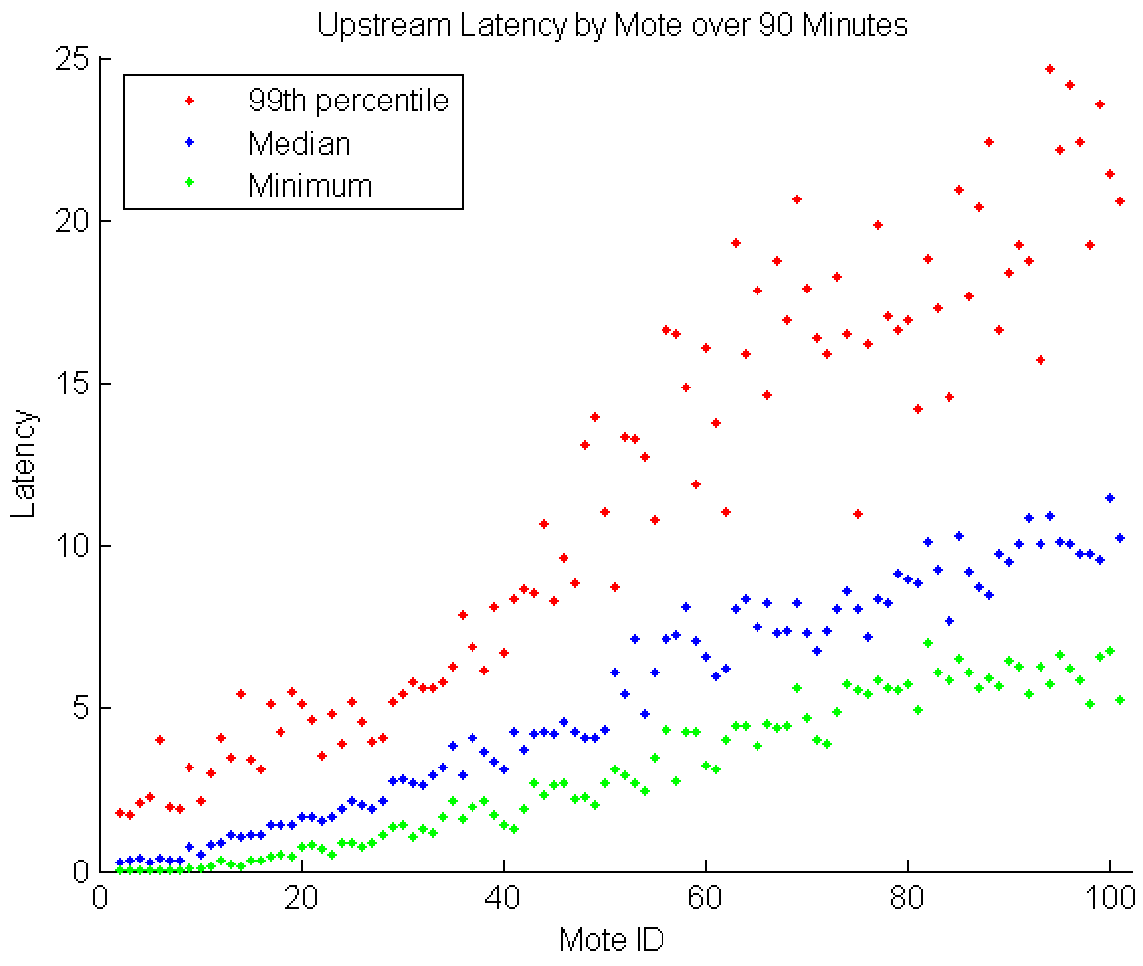


Figure 2. Latency as a function of Mote ID with the default settings over an hour of data collection. The max values are shown as red dots, the median as blue dots, and the minimum as green dots. The measured average path stability was 85% for this network.

This data can be used to provide an estimate of the latency of the deepest mote in the network. In this context, the average path stability *S* is an important parameter. *S* refers to the concept that due to a variety of RF related factors, not every transmission will get through on the first try: for example, 100% path stability means every packet gets through on the first try; 80% path stability means 8/10 packets get through on the first try, and 2/10 are resent automatically.

<median(latency)> = 0.75M/LS

The red dots in Figure 2 are an approximation of the 99th percentile latency that can be expected. Accordingly, you can expect the 99th percentile to be between 2-3 times the median latency. If path stability is low, packets may build up in the motes' queues and further increase the tails of the latency distribution. See the Application Note: Debugging Congested Networks for more discussion on identifying and mitigating the effects of full queues.

10.7 Covering Distance

There are several use cases where a long linear distance must be covered using multiple Embedded Manager networks. As an example, consider a 100km transmission line where motes are located every 200m. This line can be monitored using five 100-mote networks. In this setting, there are two alternatives:

1. If it is easy to find locations suitable for managers, put a single manager in between two linear groups of 50 motes. This has the advantage of either reducing latency by half or reducing power in the network relative to a single string of 100 motes. In this case, there are 100 motes of distance between each manager.
2. If it is difficult to find locations suitable for managers, i.e. locations with power and internet connectivity, two managers can be co-located with 100-mote networks heading off in opposite directions. In this case, there are 200 motes of distance between each pair of managers.

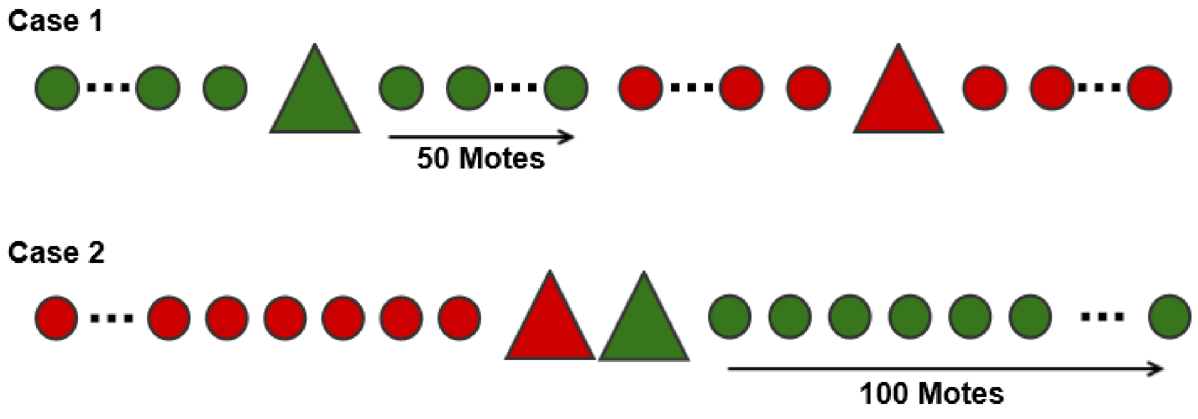


Figure 3. Alternative deployment strategies showing two networks separated by color; (top) deployment for minimizing latency or power and (bottom) deployment for minimizing number of manager sites.

To estimate performance in case 1, set M=50 in the latency and link calculations as the deepest mote is now 50 motes from the manager. However, ensure that the product $2LM < 1800$. To estimate performance in case 2, set M=100 in all the calculations.

11 Application Note: Overlapping Networks

11.1 Introduction

A deployment may require hundreds or thousands of motes in a small area spanning multiple SmartMesh IP networks. Since the SmartMesh protocol emphasizes time synchronization as a requirement for reliability, it can seem risky to deploy these motes in multiple networks located in overlapping spaces. As we will show in this Application Note, providing that the base path stability is high enough, there are no significant risks to deploying overlapping networks. We'll quantify the largest effect, which is drop in the effective path stability across all the overlapping networks. We define a single *radio space* as the area covered by the transmission from one wireless device. Saying that several devices are in the same radio space means that each device will be able to hear transmissions from any other device in this space, and also will be subject to interference from the other devices. When two transmissions occur simultaneously from two different devices in the same radio space on at the same transmission frequency, we say that these transmissions *collide*.

As a real-world example of overlapping network operation, at Dust Networks we run all of our testing in an environment where 1000 motes in 10 to 30 networks are simultaneously running within the same radio space. These networks have colliding transmissions, but there are no qualitative reductions in overall data reliability.

11.2 Method

We consider a deployment where multiple Embedded Manager SmartMesh IP networks are each given their full capacity of 100 motes. Suppose there are 10 such networks all in the same radio space, and that we pick one mote from one of the networks. Because of the SmartMesh manager's bandwidth allocation algorithms, a transmission from this mote will never collide with any other transmission from the other 99 motes in its network. However, it does have a chance at colliding with a transmission from one of the overlapping networks. Because of the decoding involved, whichever transmission started earlier is likely to be properly received and whichever transmission started midway through is likely to require a retransmission.

Based on how often motes in each network report, we can calculate the total number of transmissions per unit time for that network. And based on this total number of transmissions, we can calculate the chance of collisions with the single mote we picked out. If a collision results in our mote failing its transmission, it will automatically retry the transmission on the next assigned link. Each retry lowers the measured path stability for our mote, but data reliability is maintained since the data is simply transmitted on the next assigned link.

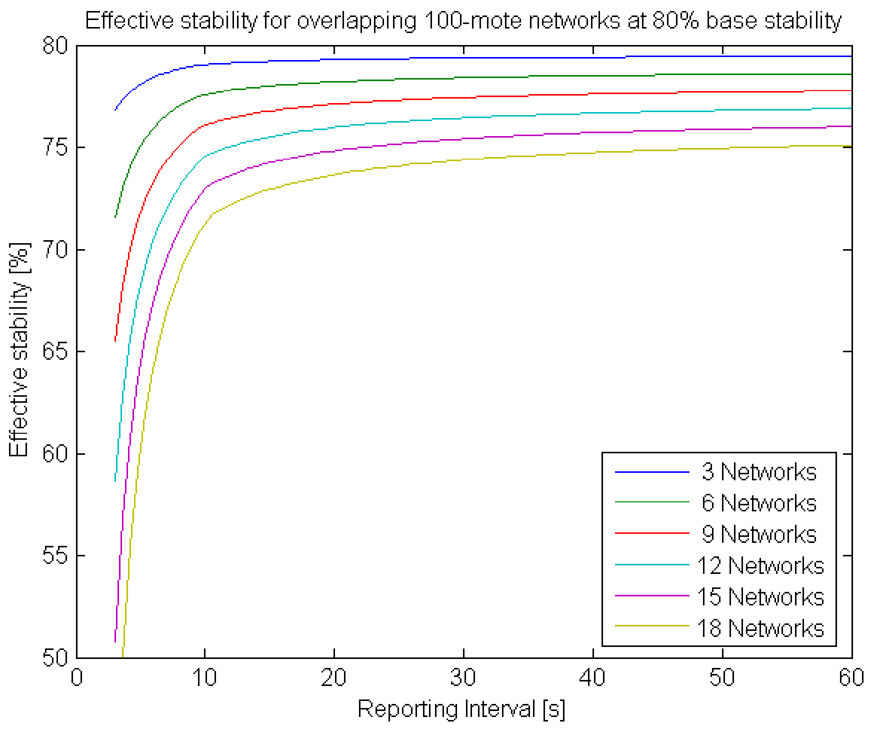
If you deployed a single 100-mote network and measured the path stability, it might be 80%. This is what we will call the *base path stability*. If we now have 10 overlapping networks, the stability is going to drop; it'll drop more the more traffic there is in the overlapping networks. We will call this final value the *effective path stability*. In the following section we will calculate the effective stability given different numbers of overlapping networks, different reporting rates, and different base stability. SmartMesh networks are designed to operate with perfect data reliability even when each transmission, on average, requires one retry to be successfully received.

In general, we do not recommend operating networks when the effective path stability is less than 50%. Networks operating in this stability regime are more prone to mote resets due to transport layer failures with the manager and can experience lower data reliability. Building networks which operate reliably in <50% stability regimes would require the SmartMesh manager to allocate bandwidth very conservatively which would limit network throughput and increase mote power.

11.3 Results

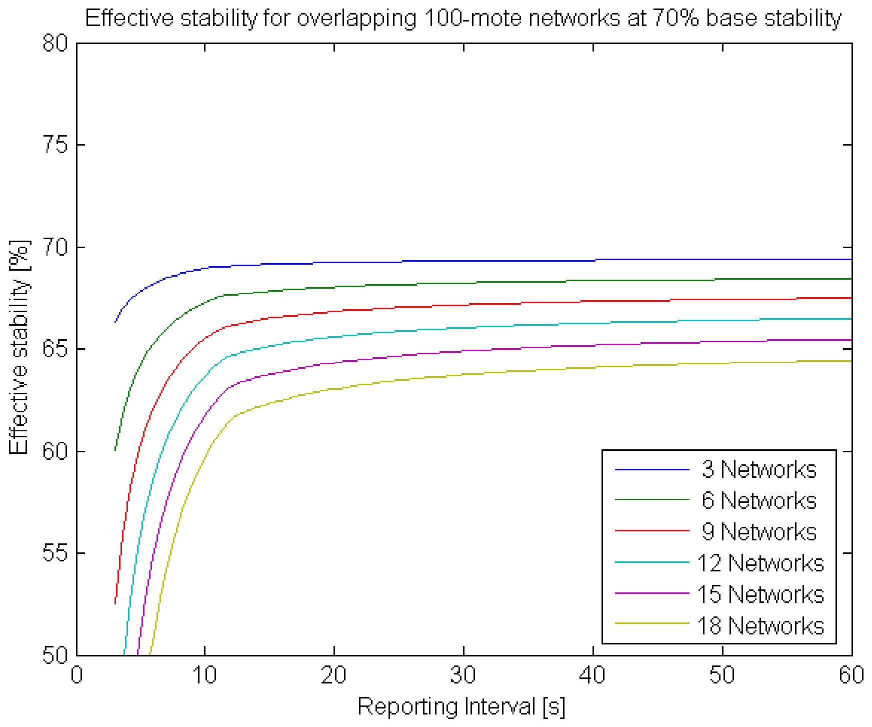
For our calculations, we assume the worst case for the deployments. First, all motes and managers are assumed to be in the same radio space. Second, all transmissions are assumed to have the maximum application-layer payload of 90 bytes so that they last as long as possible. Third, each network has a full 100 motes. We also assume that all the motes in all the overlapping networks are reporting data at the same rate. At the slow end, the motes report data on 60-second intervals. At the fast end, which is near the packet/s limit for a Embedded Manager SmartMesh IP network, motes report data on 3-second intervals.

Starting with a base stability of 80%, we see that we can tolerate many networks in the same space before stability starts to suffer.



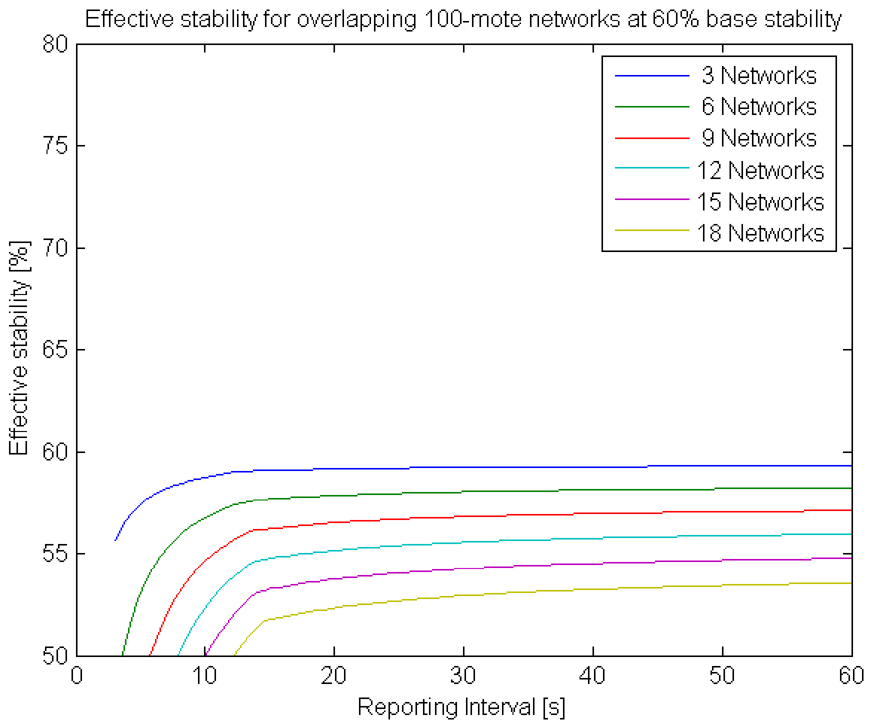
From this plot, it is apparent that if you have a base stability of 80%, you can run 15 overlapping networks at their full capacity (i.e. each mote reporting 1 packet every 3s) and not see an effective stability below 50%. There will be power and latency increases commensurate with the drop in effective stability, but reliability should be preserved.

Repeating the same analysis for 70% base stability gives us reporting limits for anything at or above 12 overlapping networks.



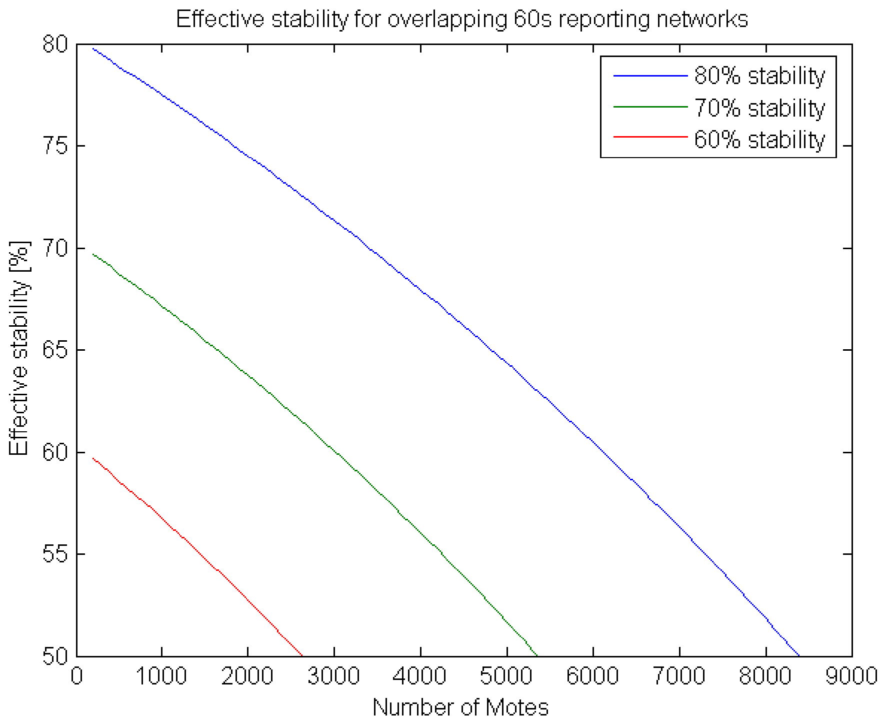
To get these limits, look at where the various curves intersect the 50% effective stability axis. It is within the recommended guidelines to run 9 networks in the same radio space with each mote reporting 1 packet every 3s.

If we next consider a base stability of 60% plotted on the same axes, there is much less margin for reliable operation within the recommended guidelines.



Here all but the 3-network case can have sub-50% effective stability.

There is one more way we can frame these results: if we have minimally reporting motes at 60s intervals, how many motes can we safely locate in the same radio space?



The family of curves for 60%, 70%, and 80% base stability is shown in the above figure. If we are starting with a high base stability, it is possible that a single radio space can support several thousands of motes. At these high mote densities it doesn't really matter how many networks are present. For example, we could have 5,000 motes as 100 networks of 50 motes each or 50 networks of 100 motes each. The same amount of traffic, and hence interference, will be present.

11.4 Conclusions

We can draw some conclusions from the plots in the previous section:

- Any installation with up to 1,800 motes spread out over 18 networks, averaging a publish interval of 1 packet per 15s, will meet the recommended guidelines. An effective path stability reduction of 5% to 10% is expected which will result in a small increase in mote energy consumption and latency. Use the SmartMesh Power and Performance Estimator to quantify these increases.
- Measuring the base stability with a test deployment of a single network of 100 motes is recommended to determining the degree of overlap and traffic level that is safe for maintaining network reliability.

To further mitigate any interference effects, these steps can be taken:

- If you are sending packets with less than full payloads, accumulate the data until a full payload is waiting at the sensor processor before sending the data to the mote. This results in higher latency but less overall radio on-time that can cause collisions. If the application can tolerate longer latency, this is a good policy for lowering mote power anyway.
- Increase the number of parents per mote from 2 to 3 - this provides more path diversity for the motes, in case some paths fail the mote will be less likely to reset. This does require more power at the parent motes, however. From the SmartMesh manager CLI, issue the commands:

```
> set config numparents=3
> reset system
```

- If the network is a low-traffic network, increase the minimum number of links per mote. This randomizes the motes' transmission schedules and prevents persistent collisions and has a similar power penalty as the additional parent above. From the SmartMesh manager CLI, issue the commands:

```
> su becareful
> seti ini minlinks=4
> reset system
```

- Any co-located Access Point devices for different networks should always be separated by at least one meter.

We do not recommend lowering the transmit power to improve overlapping network performance as this will lower all signals, including the desired ones, by the same amount. Additionally it will lower the base stability if there is any ambient RF noise in the environment.

12 Application Note: How to Read Mote Parameters Remotely

This Application Note describes the IP Mote feature that allows over-the-air access to a limited number of commands. The *netGetParameter* (id=0x02) command can be used to interrogate various mote parameters. The actual parameter identifiers and format are the same as documented in the SmartMesh IP Mote Serial API Guide.

Both request and response packets described here are sent using best-effort transport. Thus, depending on the network and RF conditions, it is possible for packets to get lost. The application should account for that by retransmitting requests if no response was received after a reasonable timeout.

12.1 Request packets

To send a remote command to a mote, applications should use the Embedded Manager *sendData* (0x2C) API command or VManager *POST /motes/m/{mac}/dataPacket* API command with the following values:

Parameter	Type	Description
macAddress	MAC_ADDR	MAC address of the destination mote.
priority	INT8U	Priority of the packet. Using low (0x00) is recommended.
srcPort	INT16U	any user port 0xF0B8-0xF0BF
dstPort	INT16U	0xF0B1 (the second manager port)
options	INT8U	0x00
data	INT8U[]	The payload data of the particular request being sent, described below

Format of the 'data' field inside the *sendData*/*dataPacket* manager API is as follows:

Command Id	Request len	Request data
1 byte	1 byte	0-n bytes

For example, the payload to send *netGetParameterCmd* for *joinDutyCycle(0x06)* parameter would be:

Command Id	Request Len	Request data
0x02 (command=netGetParameterCmd)	0x01	**paramId** 0x06

12.2 Response packets

To receive responses, the application connected to the Embedded manager must first *subscribe* to data notifications (type 0x4) or use the VManager *GET /notifications* API with a *data* filter. Responses will come as notifications and can be expected to contain the following values:

Field	Type	Enum	Description
notifType	INT8U	Notification Type	0x04 (data)
timestamp	UTC_TIME_L	none	Time that the packet was generated at the mote
macAddress	MAC_ADDR	none	MAC address of the generating mote
srcPort	INT16U	none	0xF0B1
dstPort	INT16U	none	This will be the source port used in the associated request
data	INT8U[]	none	Payload of the response

The response 'data' field of the notification will be formatted as follows. Note that the Response Data always starts with an RC byte, followed by optional response information.

Command Id	Response Len	Response Data
1 byte	1 byte	RC(1byte) || 0-n bytes of additional response data

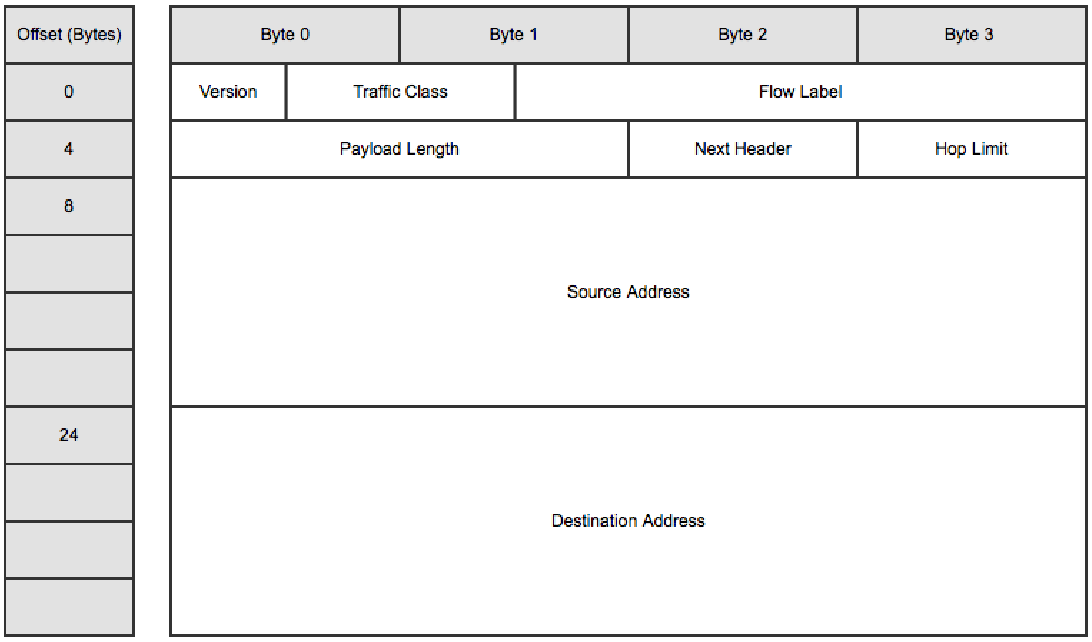
For example, the response data to retrieve *joinDutyCycle* may look like this. Here, the mote responds with *joinDutyCycle* of 0x7F.

Command Id	Response Len	Response data		
		RC	**paramId**	**joinDutyCycle**
0x02 (command=netGetParameterCmd)	3	0x00	0x06	0x7F

13 Application Note: 6LoWPAN and Routing in a SmartMesh IP Network

SmartMesh IP products use the 6LoWPAN header compression scheme as defined in RFC4944 and as updated by RFC6282 . This allows SmartMesh IP to carry IPv6 network and transport layer headers in an efficient manner, but that isn't the whole story. The IPv6 Net header consists of a 40-Byte fixed length portion, followed by a number of possible extension headers of varying lengths. The fixed header contains a number of fields related to quality of service and network management (traffic class, flow label), end-to-end addressing (source and destination IPv6 address), and a description of the next extension header (next header). 6LoWPAN compresses this header down to as little as 2 Bytes (wow!) by encoding each field to 1-3 bits which specify the level of compression, from completely elided (no additional bytes) to carrying the entire field inline (e.g. 16 Bytes for addresses).

6LoWPAN also offers efficient header compression when the next header is a UDP transport header - when using the most compressible ports, it only takes an additional Byte. What it doesn't do is offer any benefit for other Internet Protocol Suite headers, such as the TCP transport header or the ICMPv6 internet header.



13.1 Routing

There are two primary routing paradigms in mesh networks, generally referred to as "route-over" or "mesh-under."

13.1.1 Route-over

The route-over approach treats the mesh as an extension of the IP network, which could be a LAN or the Internet as a whole. This means that all nodes are IP routers of some sort, with the following implications:

- Nodes must discover their place in the network by sending ICMPv6 Neighbor Solicitation/Advertisement and Router Solicitation/Advertisement messages, none of which are efficiently compressed in existing standards (see RFC7400 for a possible solution)
- Nodes must maintain routing tables, or a simple forwarding scheme is used by creating a cost function associated with the distance each node is from the entry/exit point referred to as the root of a directed acyclic graph (the DODAG root)
- Nodes must advertise their cost function to their neighbors
- IP-layer fragmentation support is required
- Admission to the mesh network and the LAN are coupled - this has implications for security suites and their cost to implement

This approach makes the mesh look like it is part of "the internet" despite the fact that the mesh has radically different properties (speed, bandwidth) compared to a typical Ethernet or WiFi link.

13.1.2 Mesh-under

The mesh-under approach treats routing within the mesh separately from routing in the IP network, effectively flattening the mesh to one IP hop. This means that IP routing only takes place at the source and destination of the packet, with the following implications:

- There needs to be an additional routing mechanism (e.g. a header) for mesh delivery
- Neighbor discovery can use highly compressible UDP packets
- For packets flowing within the mesh, it is possible to elide most addresses, since there is already a mesh address attached to the packet
- Nodes don't need to maintain IP routing tables
- IP-layer fragmentation may not be required
- Admission to the mesh and the LAN are decoupled

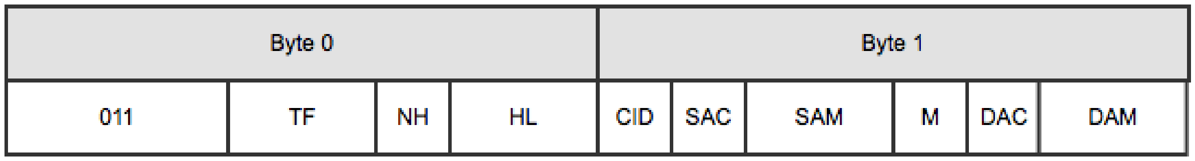
This approach requires a Low-power Border Router (LBR) to bridge between the mesh and the internet or LAN - the LBR can throttle traffic to account for the bandwidth difference between the mesh and IP networks.

13.2 SmartMesh IP 6LoWPAN Options

The SmartMesh IP Embedded Manager presents two different APIs for sending data to a mote - *sendData*, and *sendIP*. Both APIs take a mote MAC address as the packet destination - SmartMesh IP uses stateless address auto-configuration so the prefix is not needed within the mesh. The *sendData* API takes a source and destination port as arguments, and the manager constructs the 6LoWPAN header (with most of the 6LoWPAN fields elided) and sends a UDP packet to the mote. The *sendIP* API requires that the API client construct the 6LoWPAN header, so it allows for more flexibility, however only some of the possible encodings are used in the current SmartMesh IP stack.

Similarly, the VManager has *POST /motes/m/{mac}/dataPacket* and *POST /motes/m/{mac}/ipPacket* commands to send a packet to a mote, with the manager constructing the 6LoWPAN header (*dataPacket)* or the application constructing the LoWPAN header (*ipPacket*).

The 6LoWPAN header consists of a instructions on how to compress/decompress an IPv6 header :



Byte 0				Byte 1					
011	TF	NH	HL	CID	SAC	SAM	M	DAC	DAM

6LoWPAN Header

Field	Width (bits)	Description	Acceptable Values
011	3	This is a fixed field that identifies this as a reserved dispatch type	b011
TF	2	Traffic Class & Flow label - the stack foes not carry traffic class or field label	b11 = “Traffic Class and Flow Label are elided”
NH	1	Next Header	b1 = “The Next Header field is compressed and encoded using LOWPAN_NHC”
HL	2	Hop Limit (IP hops - the mesh counts as 1 hop)	b10 = "The Hop Limit field is elided and the hop limit is 64"

CID	1	Context Identifier - For traffic contained within the mesh (source and destination are both a mote or manager), the context identifier = 0. For all other traffic, the source side (LBR or mote) will assign a context and the Context Identifier bit = 1. An additional Context Identifier Extension (CIE) Byte is appended to the header. The upper nibble is source ID, and the lower nibble is destination ID.	b0 = “No additional 8-bit Context Identifier Extension is used." or b1 = "An additional 8-bit Context Identifier is used and present in the packet"
SAC	1	Source Address Compression - only certain combinations of SAC and SAM are allowed	b0 = “Source address compression uses stateless compression” or b1 = “Source address compression uses stateful, context based compression”
SAM	2	Source Address Mode	If SAC = b0, SAM = 00, “128 bits. The full address is carried in-line.“ If SAC = b1, SAM = 11, “0 bits. The address is derived using context information and possibly link-layer addresses”
M	1	Multicast Compression - use of multicast addresses always result in mesh broadcast, as there is no support for multicast groups	b0 = “Destination address is not a multicast address” or b1 = "Destination address is a multicast address" If b1, then DAC=1 and DAM=1, i.e. the multicast address is never carried inline.
DAC	1	Destination Address Compression - only certain combinations of DAC and DAM are allowed	b0 = “Destination address compression uses stateless compression” or b1 = “Destination address compression uses stateful, context based compression”

DAM	2	Destination Address Mode	If DAC = b0, DAM = 00, “128 bits. The full address is carried in-line.“ If DAC = b1, DAM = 11, “0 bits. The address is derived using context information and possibly link-layer addresses”

Summarizing:

For in-mesh traffic (both source and destination on context 0), the 6LoWPAN_IPHC header is b011.11.1.10.0.1.11.0.1.11 = 0x7E77 and the context ID Byte is elided.

13.2.1 Next Header Encoding

Next header encoding is used when an extension header is used, such as for fragmentation, ICMPv6, or TCP packets. Currently the SmartMesh IP stack only supports UDP next headers.

13.2.2 UDP Header Compression

UDP headers have a specific compression format, UDP LOWPAN_NHC, used in place of the NHC format. The UDP LOWPAN_NHC is 1B as follows:

Field	Width (bits)	Description	Acceptable Values
11110	5	This is a fixed field that identifies this as a UDP LOWPAN_NHC	b11110
C	1	Checksum - the UDP checksum can always be elided as mesh security ensures uncorrupted delivery	b1 = “2-Byte checksum is elided”

P	2	Port Compression - there is no restriction on which ports can be used for user traffic, but Linear sample applications typically use 0xF0Bx ports	b00: All 16 bits for both Source Port and Destination Port are carried in-line. This is when both ports are outside of the compressible range. b01: All 16 bits for Source Port are carried in-line. First 8 bits of Destination Port is 0xf0 and elided. The remaining 8 bits of Destination Port are carried in-line. b10: First 8 bits of Source Port are 0xf0 and elided. The remaining 8 bits of Source Port are carried in-line. All 16 bits for Destination Port are carried in-line. b11: First 12 bits of both Source Port and Destination Port are 0xF0B and elided. The remaining 4 bits for each are carried in-line.

This is followed by the UDP source/dest port list, which may be raw (2B source + 2B dest), partially compressed (2B raw source/dest + 1B compressed , i.e. 0xF0nn, source/dest), or fully compressed (1B, i.e. 0xF0Bn source/dest). Mesh control traffic uses UDP port 0xF0B0. No application traffic will be allowed on this port.

Summarizing:

For ports in the 0xF0Bx range, the compressed UDP LOWPAN_NHC header is b11110.1.11 = 0xF7, and the header itself will be 0xsd, where s= source port and d = destination port.

14 Application Note: Building a VManager OTAP Application

Mote firmware can be upgraded live in a network – this is called Over-The-Air-Programming (OTAP). This process securely downloads a firmware image onto one or more motes, replacing the existing firmware with a new version. It is a slow process, but does not interrupt normal mote function until the device resets at the end of the update.

An external OTAP application is responsible for managing the upgrade process - deciding which motes to update, what software to upgrade them to, what progress information to display, etc. are up to the OTAP application developer. The SmartMesh SDK provides a Python reference application (`VMgr_OTAPCommunicator.py`), that can be used to do OTAP, but you may wish to build your own in another language. Refer to the OTAP Communicator Documentation for details on the reference application.

Linear provides firmware updates in the form of an `.otap2` file (for LT5800-based devices). Linear also provides `OTAPfile.py` to generate `.otap2` files for custom applications built using the On-Chip Software Development Kit.

14.1 How OTAP Works

The OTAP application flow has the following steps:

1. Determine which motes to OTAP (the "receive list")
2. Initiate the OTAP process by sending an OTAP handshake to each mote being updated
3. Break up the OTAP file into blocks that will fit in a packet
4. Send the fragmented OTAP file to all the motes in the network - only those that complete the handshake will use the update
5. Monitor OTAP status on all the motes
6. Once the complete image has been received by all motes, commit the firmware to flash

An OTAP application uses the Manager's *POST /motes/m/{mac}/dataPacket* API command (the "**dataPacket**" API) to send OTAP protocol packets to the motes, where the payload field is an OTAP command (described below). The OTAP application monitors the process using OTAP command responses from the motes - the OTAP application must subscribe to data notifications (*GET /notifications*) to see OTAP responses.

The **dataPacket** API takes the MAC address of the mote as a parameter. FF-FF-FF-FF-FF-FF-FF-FF can be used to broadcast to all motes for commands that allow broadcast commands, e.g. OTAP data. The source/destination port = 0xF0B1 = 61617 for OTAP commands. The payload supplied to the send API consists of an API header and API payload - see below for details on their contents.

See the SmartMesh IP VManager API Guide for details on *POST /motes/m/{mac}/dataPacket*.

14.1.1 Prepare a "Receive List" of Motes to be OTAPed

The OTAP application can take a list of mote MAC addresses as an input, or it could query the mote version for each mote to see if they are suitable for an upgrade.To query each mote, your application must:

1. Build a list of motes by issuing *GET /motes*
2. For each mote on the list, use *GET /motes/m/{mac}/info* to get the *appSWRev* string. Only devices with a suitable version are placed on the "receive list"

This API also returns an *appId* - this is currently a reserved field, so it cannot be used to identify custom firmware.

14.1.2 Unicast Handshake with the Motes on the Receive List

For each mote on the receive list, the OTAP application will use the **dataPacket** API to send a packet containing an *OTAPHandshake* (0x16) command with a length of 34 (0x22) Bytes.

OTAPHandshake Request

Parameter	Type	Description
otapFlags	INT8U	OTAP flags = 0x07. This tells the mote that the file is an `.otap2` file, to overwrite any previous OTAP file if it exists, and to mark the file as temporary (can be deleted).
otapMIC	INT32U	Special OTAP MIC of the file payload - this is not the MIC in the `.otap2` file. See "Calculating the otapMIC" below.
fileSize	INT32U	File size = # of Bytes in the `.otap2` file. A 4-Byte size field is found at offset 0x4 in the `. otap2` file. The fileSize is the value in the field + 8 Bytes.
blkSize	INT8U	Block size = # of payload Bytes in each OTAPdata packet = 0x48 (72 Bytes - see below)
fileInfo	dn_api_otap_fileinfo_t	OTAP file information structure. See below

OTAP File Info

The OTAP File Info structure can be found in the `.otap2` file starting at offset 0x8. It will be validated against the `.otap2` file that the mote receives in the next step.

dn_api_otap_fileinfo_t Structure

Parameter	Type	Description
partition id	INT8U	Partition ID = 0x02
flags	INT8U	File Flags (bitmap): Either 0x03 or 0x07 • b0 = 1 (executable) • b1 = 1 (LZSS compressed) • b2 = 0 (validate vendor and application ID) or 1 (skip validation) Bit 2 will depend upon the settings used when generating the .otap2 file. Factory software ships with b2=0.
fileSize	INT32U	Size of uncompressed .bin image
exeStartAddr	INT32U	Start address of executable = 0x00041020
exeVersion	APP_VER	Version of the application being OTAP'd - see APP_VER description above for *getParameter* response
exeDependsVersion	APP_VER	Depends version (mote must be at this version or later to proceed)
exeAppId	INT8U	Application ID = 0x01*
exeVendorId	INT16U	Vendor ID = 0x0001*
exeHwId	INT8U	Hardware ID = 0x03

*Vendor ID and Application ID are currently reserved fields and must be set as indicated in the table.

If a mote accepts the handshake it will return an acknowledgement containing an API response code, an OTAP response code, the *otapMIC* of the file from the request, and *genDelay* which is the required delay (in ms) between packets for that mote to meet its power target - the OTAP application should space packets by the largest delay returned by all the motes on the receive list or one downstream frame (~ 2s default for the embedded manager), whichever is larger. Only the response code is returned if the response code is not RC_OK.

OTAPHandshake Response

Parameter	Type	Description
rc	INT8U	API response code

otapRc	INT8U	Otap-specific return code (see below)
otapMIC	INT32U	The otapMIC given in the *OTAPHandshake* request
genDelay	INT32U	Requested delay between packets (in ms)

14.1.3 Calculating the otapMIC

A 4-Byte *otapMIC* is calculated over the entire file (the plaintext) by computing the 128-bit AES CBC output. This can be done inside your OTAP application, or out-of-band (e.g. using openssl) and presented as an input to the OTAP application along with the mote binary. It takes a 16-Byte key and 16-Byte nonce (the initialization vector) as inputs in addition to the data being MIC'd.

- key = `c3 bd 8f 3c c7 c9 99 29 22 92 f3 f2 a2 9d c3 10`
- nonce = `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`

The file is extended to a multiple of the 16-Byte block size by padding with zeros if necessary.

The *otapMIC* is the first 4 Bytes of the last output block in the chain.

Example

Data (27 Bytes) = `81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f 90`

`71 72 73 74 75 76 77 78 79 7a 7b`

Padded data (32 Bytes) = `81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f 90`

`71 72 73 74 75 76 77 78 79 7a 7b 00 00 00 00 00`

encoded data (32 Bytes) = `4c f9 4f 26 a3 2d c2 5b 81 40 0b c5 30 d1 a7 e0`

`c5 a2 09 34 05 4b 7f b0 da 56 5d 89 53 c3 a8 fd`

otapMIC (4 Bytes) = `c5 a2 09 34`

14.1.4 Fragment the .otap2 File into Network Packets

The OTAP application must break the entire `.otap2` file into data blocks that will fit into a downstream broadcast packet. The maximum block size (the payload field in the *OTAPData* request below is 72 (0x48) Bytes. All data blocks must be the same size specified in the *OTAPHandshake*, however it is not necessary to pad the last block to meet the block size.

14.1.5 Broadcast the OTAP Data Blocks to the Network

OTAP consists of multiple rounds. In the first round, the OTAP application sends each data block in the `.otap2` file to the broadcast address (FF-FF-FF-FF-FF-FF-FF-FF) using the **dataPacket** API containing an *OTAPData* (0x17) command with a length of 78 (0x4E) Bytes. Motes do not respond to each *OTAPData* packet. Only devices on the receive list that accepted the handshake will do anything with the file.

OTAPData Request

Parameter	Type	Description
otapMIC	INT32U	Must match the otapMIC given in the *OTAPHandshake*
block	INT16U	Block number
payload	INT8U[]	The data block payload (72 Bytes)

14.1.6 Monitor Update Status

The OTAP application can check on the status of the update by sending a command to all motes on the receive list asking which blocks were not received using the **dataPacket** API containing an *OTAPStatus* (0x18) command with a length of 4 (0x04) Bytes. This can be done by broadcasting to the entire network, or sending directly (unicasting) to each mote on the receive list in sequence. In either case, if a mote fails to respond, the OTAP application should retry the *OTAPStatus* command.

OTAPStatus Request

Parameter	Type	Description
otapMIC	INT32U	Must match the otapMIC given in the *OTAPHandshake*

The motes will return an acknowledgement containing an API response code, an OTAP response code, the *otapMIC* of the file from the request, and a list of blocks it did not receive. Only the response code is returned if the response code is not RC_OK.

OTAPStatus Response

Parameter	Type	Description
rc	INT8U	API response code
otapRc	INT8U	Otap-specific return code (see below)
otapMIC	INT32U	The otapMIC given in the *OTAPStatus* request
lostBlocks	INT16U[]	List of missing blocks

Once all the motes on the receive list have responded to the *OTAPStatus* command, a list of all the missed data blocks is created. The list is then used in the next round and the process is repeated.

The mote can only report a maximum of 40 lost blocks, which is 1-2% of the total number of blocks. Normally this is not a problem, but in cases of marginal network stability, it may result in the application constructing an incomplete list of missing blocks based on the first *OTAPStatus* check. Either the application can retry the blocks it knows about, then re-check the status, which will return additional blocks; or the application could get the OTAP status periodically after a smaller number of blocks, such that the network wide list of missing blocks is likely to be complete at the end of the first round.

When all motes on the receive list indicate that they have received all data blocks (this typically takes several rounds), the file is ready to be programmed onto the motes.

The OTAP application can display progress information in several ways, e.g.:

- Display current round, number of blocks sent, and number of blocks reported received (per-mote or average) at end of last round
- Display a simple % of done (average) for the network

14.1.7 Unicast Commit Commands to the Receive List

Firmware is programmed using the **dataPacket** API containing an *OTAPCommit* (0x19) command with a length of 4 (0x4) Bytes. The OTAP application sends the command to each mote on the receive list that indicated it had received all data blocks. The command tells the motes to reprogram their flash with the new firmware and reset.

OTAPCommit Request

Parameter	Type	Description
otapMIC	INT32U	Must match the otapMIC given in the *OTAPHandshake*

The motes will return an acknowledgement containing an API response code, an OTAP response code, the *otapMIC* of the file from the request, and a commit response code. Only the response code is returned if the response code is not RC_OK.

OTAPCommit Response

Parameter	Type	Description
rc	INT8U	API response code
otapRc	INT8U	OTAP response code (see below)
otapMIC	INT32U	The otapMIC given in the *OTAPCommit* request

After sending the *OTAPCommit* response, the mote will reset, reprogram itself, reset again, then resume operation.

14.1.8 Motes Reset and Rejoin

Once OTAP has completed, the OTAP application can show the result of OTAP in several ways, e.g.:

- Display a table of the motes on the receive list and whether or not they rejoined after OTAP
- Extend this table by querying and displaying the new firmware version
- Print a success message and terminate

14.2 Q&A

- Q. What happens if you do a commit before all blocks are received?
  - A. The mote will return a DN_API_OTAP_RC_MIC error and abort OTAP. OTAP must be restarted on the mote that didn't receive all the blocks. This situation may occur if the OTAP application is designed to time out after a fixed number of rounds.

- Q. What happens if a mote resets mid-OTAP?
  - A. OTAP is aborted on that mote. OTAP will need to be restarted on that mote. For example, if a mote resets after 500 blocks have been sent, then the OTAP application can send an *OTAPHandshake* to that mote, and continue sending *OTAPData* messages, but it will have to repeat the first 500 blocks in subsequent rounds.

- Q. What happens if the `.otap2` file doesn't verify (e.g. doesn't pass version dependence)?
  - A. The mote will return an error during *OTAPCommit.*

- Q. What if a mote doesn't respond to a command?
  - A. The manager will retry the *OTAPHandshake*, and *OTAPCommit* commands until a routing timeout occurs (can be several minutes). Your application can safely retry these commands if a response isn't received within 30 s. The mote may respond with an error if the original command was received, but the response has not reached the manager. The *OTAPData* (and possibly *OTAPStatus)* commands are broadcast once. Your application can safely repeat both commands 2 or 3 times to marginally increase the likelihood of a block being delivered, however this slows down the process, so it is usually better to try once and resend in a subsequent round as needed.

- Q. What if I OTAP a version that requires a different loader (e.g. 1.0.5.4) than is currently on the part (e.g. 1.3.0.12)?
    - A. The *OTAPCommit* will appear to succeed, since the mote compares the *otapMIC* in the file to that in the *OTAPHandshake*, but when the part is rebooted, the loader will not be able to verify the *otapMIC*, since the signature algorithm changed between loader 1.0.3 and 1.0.5 for FIPS 140-2 compliance. The loader will need to reprogrammed using the Eterna Serial Programmer software - this can be done in the field provided that the motes SPI programming interface is available in order to accept the new firmware.

- Q. Can I speed up OTAP?
    - A. You can't speed up the rate at which packets flow into the network without changing the downstream frame size. This is not possible in the embedded manager, but is in VManager. It is usually faster to wait for motes that haven't received a few blocks to "catch up" than it is to do an *OTAPCommit* and restart OTAP with only the failed motes on the receive list.

- Q. I used `OTAPFile.py` to generate an `.otap2` file, and my OTAP application said it programmed with no errors, but the motes are still using the old version. What happened?
    - A. On-chip SDK versions prior to 1.1.0.8 included an `OTAPFile.py` file that used the wrong signature algorithm to generate the *otapMIC*. Upgrade to the newer version.

14.3 OTAP Response Codes

The following response codes may be returned by the OTAP commands:

Name	Value	Description
DN_API_OTAP_RC_OK	0	Command accepted
DN_API_OTAP_RC_LOWBATT	1	Battery voltage too low to write flash
DN_API_OTAP_RC_FILE	2	File size, block size, start address, or execute size is wrong
DN_API_OTAP_RC_INVALID_PARTITION	3	Invalid partition information (ID, file size)
DN_API_OTAP_RC_INVALID_APP_ID	4	AppId is not correct
DN_API_OTAP_RC_INVALID_VER	5	SW versions are not compatible for OTAP
DN_API_OTAP_RC_INVALID_VENDOR_ID	6	Invalid vendor ID
DN_API_OTAP_RC_RCV_ERROR	7	
DN_API_OTAP_RC_FLASH	8	Other flash error
DN_API_OTAP_RC_MIC	9	MIC failed on uploaded data
DN_API_OTAP_RC_NOT_IN_OTAP	10	No OTAP handshake is initiated
DN_API_OTAP_RC_IOERR	11	IO error
DN_API_OTAP_RC_CREATE	12	Cannot create OTAP file
DN_API_OTAP_RC_INVALID_EXEPAR_HDR	13	Wrong value for exe-partition header fields 'signature' or 'upgrade'
DN_API_OTAP_RC_RAM	14	Can not allocate memory
DN_API_OTAP_RC_UNCOMPRESS	15	Decompression error
DN_API_OTAP_IN_PROGRESS	16	OTAP already in progress
DN_API_OTAP_LOCK	17	OTAP is locked out

15 Application Note: Building an Embedded Manager OTAP Application

Mote firmware can be upgraded live in a network – this is called Over-The-Air-Programming (OTAP). This process securely downloads a firmware image onto one or more motes, replacing the existing firmware with a new version. It is a slow process, but does not interrupt normal mote function until the device resets at the end of the update.

An external OTAP application is responsible for managing the upgrade process - deciding which motes to update, what software to upgrade them to, what progress information to display, etc. are up to the OTAP application developer. The SmartMesh SDK provides a Python reference application (`OTAPCommunicator.py`), that can be used to do OTAP, but you may wish to build your own in another language. Refer to the OTAP Communicator Documentation for details on the reference application.

Linear provides firmware updates in the form of an `.otap2` file (for LT5800-based devices). Linear also provides `OTAPfile.py` to generate `.otap2` files for custom applications built using the On-Chip Software Development Kit.

15.1 How OTAP Works

The OTAP application flow has the following steps:

1. Determine which motes to OTAP (the "receive list")
2. Initiate the OTAP process by sending the OTAP handshake to each mote being updated
3. Break up the OTAP file into blocks that will fit in a packet
4. Send the fragmented OTAP file to all the motes in the network - only those that complete the handshake will use the update
5. Monitor OTAP status on all the motes
6. Once the complete image has been received by all motes, commit the firmware to flash

An OTAP application uses the Manager's *sendData* API command to send OTAP protocol packets to the motes, where the payload is an OTAP command (described below). The OTAP application monitors the process using OTAP command responses from the motes - the OTAP application must subscribe to data notifications to see OTAP responses.

SendData Request

Parameter	Type	Enum	Description
macAddress	MAC_ADDR		MAC address of the destination mote; FF-FF-FF-FF-FF-FF-FF-FF can be used to broadcast to all motes
priority	INT8U	Packet Priority	Priority of the packet; Low should be used for OTAP

srcPort	INT16U		Source port
dstPort	INT16U		Destination port
options	INT8U		The options field is reserved for future use; it must be set to 0
payload	INT8U[]		API header and API payload (see below)

See the SmartMesh IP Embedded Manager API Guide for details on *sendData*. The source/destination port = 0xF0B8 = 61624 for regular API commands like *getParameter*. The source/destination port = 0xF0B1 = 61617 for OTAP commands.

15.1.1 Prepare a "Receive List" of Motes to be OTAPed

The OTAP application could take a list of mote MAC addresses as an input, or it could query the mote version for each mote to see if they are suitable for an upgrade.To query each mote, your application must:

1. Build a list of motes by issuing a *getMoteConfig* API command with a MAC address of all 00's and the *next* parameter set to true, then use the returned MAC to repeat the process until *getMoteConfig* returns RC_END_OF_LIST.
2. For each mote on the list, use the *sendData* command to remotely issue the mote's *getParameter<appInfo>* API command. Remote commands use a modified version of the API header, not HDLC encoded and no flag byte:

API Command

API Header	API Payload
Command + Length	Response code (responses only) + Message Payload

For the *getParameter* command, the API header is:

API Header for sendData

Command	Length
02 (getParameter)	1

And the API payload is the body of a *getParameter* request which contains the *appInfo* parameter ID:

getParameter Request

Parameter	Type	Description
paramId	INT8U	Parameter ID = appInfo = 0x1E

Thus the payload field in the *sendData* request would be `02 01 1E`.

The mote will return the following information about the mote in the 10-Byte API response payload that follows after the API header (here `02 0a`). Only the response code is returned if the response code is not RC_OK.

getParameter Response

Parameter	Type	Description
rc	INT8U	Response code
paramId	INT8U	Parameter ID = appInfo = 0x1E
vendorId	INT16U	Vendor ID = 0001*
appId	INT8U	Application ID = 01*
appVer	APP_VER	Application version. The serialized format is as follows: • INT8U - major - the major version • INT8U - minor - the minor version • INT8U - patch - the patch version • INT16U - build - the build version

*Vendor ID and Application ID are currently reserved fields. They may be used in future versions to identify custom firmware.

See the SmartMesh IP Mote Serial API Guide for details on the *getParameter* API command and definitions of enumerated types.

15.1.2 Unicast Handshake with the Motes on the Receive List

For each mote on the receive list, the OTAP application will issue a *sendData* command containing a *OTAPHandshake* (0x16) command with a length of 34 (0x22) Bytes.

OTAPHandshake Request

Parameter	Type	Description
otapFlags	INT8U	OTAP flags = 0x07. This tells the mote that the file is an `.otap2` file, to overwrite any previous OTAP file if it exists, and to mark the file as temporary (can be deleted).
otapMIC	INT32U	Special OTAP MIC of the file payload - this is <u>not</u> the MIC in the `.otap2` file. See "Calculating the otapMIC" below.
fileSize	INT32U	File size = # of Bytes in the `.otap2` file. A 4-Byte size field is found at offset 0x4 in the `.otap2` file. The fileSize is the value in the field + 8 Bytes.
blkSize	INT8U	Block size = # of payload Bytes in each OTAPdata packet = 0x48 (72 Bytes - see below)
fileInfo	dn_api_otap_fileinfo_t	OTAP file information structure. See below

OTAP File Info

The OTAP File Info structure can be found in the `.otap2` file starting at offset 0x8. It will be validated against the `.otap2` file that the mote receives in the next step.

dn_api_otap_fileinfo_t Structure

Parameter	Type	Description
partition id	INT8U	Partition ID = 0x02
flags	INT8U	File Flags (bitmap): Either 0x03 or 0x07 • b0 = 1 (executable) • b1 = 1 (LZSS compressed) • b2 = 0 (validate vendor and application ID) or 1 (skip validation) Bit 2 will depend upon the settings used when generating the .otap2 file. Factory software ships with b2=0.
fileSize	INT32U	Size of uncompressed .bin image
exeStartAddr	INT32U	Start address of executable = 0x00041020
exeVersion	APP_VER	Version of the application being OTAP'd - see APP_VER description above for *getParameter* response
exeDependsVersion	APP_VER	Depends version (mote must be at this version or later to proceed)
exeAppId	INT8U	Application ID = 0x01*
exeVendorId	INT16U	Vendor ID = 0x0001*
exeHwId	INT8U	Hardware ID = 0x03

*Vendor ID and Application ID are currently reserved fields. They may be used in future versions to identify custom firmware.

If a mote accepts the handshake it will return an acknowledgement containing an API response code, an OTAP response code, the *otapMIC* of the file from the request, and *genDelay* which is the required delay (in ms) between packets for that mote to meet its power target - the OTAP application should space packets by the largest delay returned by all the motes on the receive list or one downstream frame (~ 2s default for the embedded manager), whichever is larger. Only the response code is returned if the response code is not RC_OK.

OTAPHandshake Response

Parameter	Type	Description
rc	INT8U	API response code

otapRc	INT8U	Otap-specific return code (see below)
otapMIC	INT32U	The otapMIC given in the *OTAPHandshake* request
genDelay	INT32U	Requested delay between packets (in ms)

15.1.3 Calculating the otapMIC

A 4-Byte otapMIC is calculated over the entire file (the plaintext) by computing the 128-bit AES CBC output. This can be done inside your OTAP application, or out-of-band (e.g. using openssl) and presented as an input to the OTAP application along with the mote binary. It takes a 16-Byte key and 16-Byte nonce (the initialization vector) as inputs in addition to the data being MIC'd.

- key = `c3 bd 8f 3c c7 c9 99 29 22 92 f3 f2 a2 9d c3 10`
- nonce = `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`

The file is extended to a multiple of the 16-Byte block size by padding with zeros if necessary.

The otapMIC is the first 4 Bytes of the last output block in the chain.

Example

Data (27 Bytes) = `81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f 90`

`71 72 73 74 75 76 77 78 79 7a 7b`

Padded data (32 Bytes) = `81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f 90`

`71 72 73 74 75 76 77 78 79 7a 7b 00 00 00 00 00`

encoded data (32 Bytes) = `4c f9 4f 26 a3 2d c2 5b 81 40 0b c5 30 d1 a7 e0`

`c5 a2 09 34 05 4b 7f b0 da 56 5d 89 53 c3 a8 fd`

otapMIC (4 Bytes) = `c5 a2 09 34`

15.1.4 Fragment the .otap2 File into Network Packets

The OTAP application must break the entire `.otap2` file into data blocks that will fit into a downstream broadcast packet. The maximum block size (the payload field in the *OTAPData* request below is 72 (0x48) Bytes. All data blocks must be the same size specified in the *OTAPHandshake*, however it is not necessary to pad the last block to meet the block size.

15.1.5 Broadcast the OTAP Data Blocks to the Network

OTAP consists of multiple rounds. In the first round, the OTAP application sends each data block in the `.otap2` file to the broadcast address (FF-FF-FF-FF-FF-FF-FF-FF) using the *OTAPData* (0x17) command with a length of 78 (0x4E) Bytes. Motes do not respond to each *OTAPData* packet. Only devices on the receive list that accepted the handshake will do anything with the file.

OTAPData Request

Parameter	Type	Description
otapMIC	INT32U	Must match the otapMIC given in the *OTAPHandshake*
block	INT16U	Block number
payload	INT8U[]	The data block payload (72 Bytes)

15.1.6 Monitor Update Status

The OTAP application can check on the status of the update by sending a command to all motes on the receive list asking which blocks were not received using the *OTAPStatus* (0x18) command with a length of 4 (0x04) Bytes. This can be done by broadcasting to the entire network, or sending directly (unicasting) to each mote on the receive list in sequence. In either case, if a mote fails to respond, the OTAP application should retry the OTAPStatus command.

OTAPStatus Request

Parameter	Type	Description
otapMIC	INT32U	Must match the otapMIC given in the *OTAPHandshake*

The motes will return an acknowledgement containing an API response code, an OTAP response code, the *otapMIC* of the file from the request, and a list of blocks it did not receive. Only the response code is returned if the response code is not RC_OK.

OTAPStatus Response

Parameter	Type	Description
rc	INT8U	API response code
otapRc	INT8U	Otap-specific return code (see below)
otapMIC	INT32U	The otapMIC given in the *OTAPStatus* request
lostBlocks	INT16U[]	List of missing blocks

Once all the motes on the receive list have responded to the *OTAPStatus* command, a list of all the missed data blocks is created. The list is then used in the next round and the process is repeated.

The mote can only report a maximum of 40 lost blocks, which is 1-2% of the total number of blocks. Normally this is not a problem, but in cases of marginal network stability, it may result in the application constructing an incomplete list of missing blocks based on the first *OTAPStatus* check. Either the application can retry the blocks it knows about, then re-check the status, which will return additional blocks; or the application could get the OTAP status periodically after a smaller number of blocks, such that the network wide list of missing blocks is likely to be complete at the end of the first round.

When all motes on the receive list indicate that they have received all data blocks (this typically takes several rounds), the file is ready to be programmed onto the motes.

The OTAP application can display progress information in several ways, e.g.:

- Display current round, number of blocks sent, and number of blocks reported received (per-mote or average) at end of last round
- Display a simple % of done (average) for the network

15.1.7 Unicast Commit Commands to the Receive List

Firmware is programmed using the *OTAPCommit* (0x19) command with a length of 4 (0x4) Bytes. The OTAP application sends the command to each mote on the receive list that indicated it had received all data blocks. The command tells the motes to reprogram their flash with the new firmware and reset.

OTAPCommit Request

Parameter	Type	Description
otapMIC	INT32U	Must match the otapMIC given in the *OTAPHandshake*

The motes will return an acknowledgement containing an API response code, an OTAP response code, the *otapMIC* of the file from the request, and a commit response code. Only the response code is returned if the response code is not RC_OK.

OTAPCommit Response

Parameter	Type	Description
rc	INT8U	API response code
otapRc	INT8U	OTAP response code (see below)
otapMIC	INT32U	The otapMIC given in the *OTAPCommit* request

After sending the OTAPCommit response, the mote will reset, reprogram itself, reset again, then resume operation.

15.1.8 Motes Reset and Rejoin

Once OTAP has completed, the OTAP application can show the result of OTAP in several ways, e.g.:

- Display a table of the motes on the receive list and whether or not they rejoined after OTAP
- Extend this table by querying and displaying the new firmware version
- Print a success message and terminate

15.2 Q&A

- Q. What happens if you do a commit before all blocks are received?
    - A. The mote will return a DN_API_OTAP_RC_MIC error and abort OTAP. OTAP must be restarted on the mote that didn't receive all the blocks. This situation may occur if the OTAP application is designed to time out after a fixed number of rounds.

- Q. What happens if a mote resets mid-OTAP?
    - A. OTAP is aborted on that mote. OTAP will need to be restarted on that mote. For example, if a mote resets after 500 blocks have been sent, then the OTAP application can send an *OTAPHandshake* to that mote, and continue sending *OTAPData* messages, but it will have to repeat the first 500 blocks in subsequent rounds.

- Q. What happens if the `.otap2` file doesn't verify (e.g. doesn't pass version dependence)?
    - A. The mote will return an error during *OTAPCommit.*

- Q. What if a mote doesn't respond to a command?
    - A. The manager will retry the *OTAPHandshake*, and *OTAPCommit* commands until a routing timeout occurs (can be several minutes). Your application can safely retry these commands if a response isn't received within 30 s. The mote may respond with an error if the original command was received, but the response has not reached the manager. The *OTAPData* (and possibly *OTAPStatus)* commands are broadcast once. Your application can safely repeat both commands 2 or 3 times to marginally increase the likelihood of a block being delivered, however this slows down the process, so it is usually better to try once and resend in a subsequent round as needed.

- Q. What if I OTAP a version that requires a different loader (e.g. 1.0.5.4) than is currently on the part (e.g. 1.3.0.12)?
    - A. The *OTAPCommit* will appear to succeed, since the mote compares the *otapMIC* in the file to that in the *OTAPHandshake*, but when the part is rebooted, the loader will not be able to verify the *otapMIC*, since the signature algorithm changed between loader 1.0.3 and 1.0.5 for FIPS 140-2 compliance. The loader will need to reprogrammed using the Eterna Serial Programmer software - this can be done in the field provided that the motes SPI programming interface is available in order to accept the new firmware.

- Q. Can I speed up OTAP?
    - A. You can't speed up the rate at which packets flow into the network without changing the downstream frame size. This is not possible in the embedded manager, but is in VManager. It is usually faster to wait for motes that haven't received a few blocks to "catch up" than it is to do an *OTAPCommit* and restart OTAP with only the failed motes on the receive list.

- Q. I used `OTAPFile.py` to generate an `.otap2` file, and my OTAP application said it programmed with no errors, but the motes are still using the old version. What happened?
    - A. On-chip SDK versions prior to 1.1.0.8 included an `OTAPFile.py` file that used the wrong signature algorithm to generate the *otapMIC*. Upgrade to the newer version.

15.3 OTAP Response Codes

The following response codes may be returned by the OTAP commands:

Name	Value	Description
DN_API_OTAP_RC_OK	0	Command accepted
DN_API_OTAP_RC_LOWBATT	1	Battery voltage too low to write flash
DN_API_OTAP_RC_FILE	2	File size, block size, start address, or execute size is wrong
DN_API_OTAP_RC_INVALID_PARTITION	3	Invalid partition information (ID, file size)
DN_API_OTAP_RC_INVALID_APP_ID	4	AppId is not correct
DN_API_OTAP_RC_INVALID_VER	5	SW versions are not compatible for OTAP
DN_API_OTAP_RC_INVALID_VENDOR_ID	6	Invalid vendor ID
DN_API_OTAP_RC_RCV_ERROR	7	
DN_API_OTAP_RC_FLASH	8	Other flash error
DN_API_OTAP_RC_MIC	9	MIC failed on uploaded data
DN_API_OTAP_RC_NOT_IN_OTAP	10	No OTAP handshake is initiated
DN_API_OTAP_RC_IOERR	11	IO error
DN_API_OTAP_RC_CREATE	12	Cannot create OTAP file
DN_API_OTAP_RC_INVALID_EXEPAR_HDR	13	Wrong value for exe-partition header fields 'signature' or 'upgrade'
DN_API_OTAP_RC_RAM	14	Can not allocate memory
DN_API_OTAP_RC_UNCOMPRESS	15	Decompression error
DN_API_OTAP_IN_PROGRESS	16	OTAP already in progress
DN_API_OTAP_LOCK	17	OTAP is locked out

16 Application Note: Planning A Deployment

16.1 Estimating Range

Hardware integration choices influence how well devices can communicate with each other over a distance with antenna choice being the most obvious. Post-integration, device placement can change the effective range over orders of magnitude. At one end of the spectrum, devices placed on elevated poles or towers with clear line of sight to other motes in the network may have a range of 1000 m or more. At the other end, devices placed on the ground or next to large metal objects may have an effective range of 10 m or less. So when a customer asks you "What's the range of your radios", in some ways that is a meaningless or unanswerable question. You can refer to the datasheet for transmit power and receive sensitivity and the resulting link budget, but the customer will determine the range with choices they make in the development of their products and an evaluation in a real environment similar to their expected deployments.

We recommend that customers at the beginning of development plan on their devices working at a spacing of 50 m. Analysis of the first several 'typical' deployments can guide the typical range number up or down. Deployment planning simply requires that each mote, is being placed within this range of at least three existing devices. In order to form a reliable mesh, every device must have multiple neighbors and hence numerous opportunities to connect. Placing motes within range of only one other device along a maximally spaced string will result in a fragile network prone to mote resets and data loss.

16.2 Mapping out a Deployment

Once you have settled on a range for your environment, you can use a scale map to place motes at all the required sense points for the network. If possible, the AP should be located near the middle of the distribution of motes to reduce latency and mote power. Mark the AP location. Supposing that the range estimated above is 50 m, draw a circle with a 50 m radius around the manager. Not all motes within this circle will be able to communicate directly with the AP, but some motes outside the circle will, so on average it will balance out. The number of motes inside this circle approximates the number of 1-hop motes in the deployment.

Next draw a 100 m radius circle centered at the AP. The number of motes in the ring between 50 m and 100 m approximates the number of 2-hop motes. Repeat this process with circles of increasing radius until all motes have been encircled and note how many motes are in each hop. We'll use these hop counts in a minute.

There are two more things to check:

- Each mote, including the AP, should be within the estimated range of 3 other devices.
- The network should be no more than 8 hops. Deeper networks are indeed possible and should just work, but they are harder to model.

16.3 Estimating Power and Latency

Dust has provided a SmartMesh Power and Performance Estimator that estimates network performance for both product families. It can be used to estimate battery size for a given topology, packet rate, per hop latencies, and desired lifetime.

The inputs are:

- Number of motes at each hop
- Reporting interval and packet size
- Network configuration
- Path stability

With these inputs, the estimator first evaluates whether the network will form, and if so, gives:

- Mote average current by hop
- Mean latency by hop
- Network join time
- Joining mote current

We recommend using the example configurations provided to guide customer thinking on power budgeting.

There are many factors that influence power consumption and latency in actual deployments, including path stability (which changes over time) and the connectivity of the network as deployed (how many hops deep it is). The estimator provides reasonable average power for motes at a given hop - there may be motes with 3x the hop average. These estimates are for expectation setting - actual in-network performance will vary.

17 Application Note: Predicting Embedded Manager Network Health Using the CLI

Evaluating the health of a deployed and running network is important to ensure that long-term performance targets are achievable. Network health verification is simple and based on interpreting information readily available through the Embedded Manager CLI. When a network fails this network health verification test, adding more motes in key locations will usually remedy the problem.

17.1 Motivation

Once a network has been deployed and is running, it is important to verify that the network is healthy, and more importantly, that it will be healthy in the future. The network collects all the required diagnostic data for you, and it is presented at the software interfaces of the manager. It is important that early on in the development of products that the developer integrating software to the manager knows that diagnostics are important. Building the tools to automate the health verification of networks is an excellent investment that will instill confidence in the minds of end users, particularly those skeptical about wireless - see the "Monitoring SmartMesh IP Network Health" application note for details on programmatic health monitoring.

It is also possible to make this evaluation manually. The process described below will let the user know if it is safe to walk away from a network (a "green light"), and also serves to identify the source of problems in the rare case that a properly installed network does have problems.

17.2 Overview

Verifying a network involves answering two multi-part questions:

1. Does the network LOOK GOOD?
2. Does the network have the building blocks to BE GOOD?

The desired answer to both these questions is YES. If the answer is YES to both these questions, the test has passed and the network in question should be expected to run well for the foreseeable future.

17.3 Does the Network LOOK GOOD?

This part of the network evaluation test involves answering three very simple observational questions about the network. They are:

1) Is the data reliability high?

In any good deployment, the data delivery rate in the network will be close to 100%. Dust networks are built with very few mechanisms for losing data, and data reliability of >99.99% is expected. The total number of Lost and Arrived packets is printed using the `show stat` command. Confirm that 1-Lost/Arrived > 99.99%. In the example below, there are no Lost packets and 8279 Arrived packets, so reliability is 100%.

```
> show stat
Manager Statistics --------------------------------
   established connections: 1
   dropped connections    : 0
   transmit OK            : 0
   transmit error         : 0
   transmit repeat        : 0
   receive  OK            : 313
   receive  error         : 0
   acknowledge delay avrg : 0 msec
   acknowledge delay max  : 0 msec
Network Statistics --------------------------------
   reliability:  100% (Arrived/Lost:   8279/0)
   stability:     90% (Transmit/Fails: 22012/2175)
   latency:      100 msec
```

2) Is the joining behavior correct?

The first part of this question is: did all your devices join? Only the installer can know for sure how many devices were deployed. They must make sure that if they put 100 devices out there, that 100 devices joined. The second part of this question is making sure that all devices joined precisely once. If a device joined more than once, has it been continuously live in the network long enough to make you confident it is not constantly dropping out and rejoining? A device that dropped out and rejoined once while the network was building is not as worrisome as one that resets long after building is complete. All motes in the network are shown with their MAC addresses using the `sm` command, and a column shows the count of how many times each device has joined.

```
> sm
     MAC                 MoteId  State Nbrs Links Joins    Age StateTime
00-17-0D-00-00-30-0A-F4    1      Oper    6   114     1      0    0-01:53:26
00-17-0D-00-00-3F-FC-04    2      Oper    8    39     1    188    0-01:53:08
00-17-0D-00-00-38-16-2A    3      Oper    4    31     1    178    0-01:52:55
00-17-0D-00-00-3F-FC-18    4      Oper    3    58     1    167    0-01:52:47
00-17-0D-00-00-38-12-84    5      Oper    2    11     1    163    0-01:52:42
00-17-0D-00-00-3F-FD-3B    6      Oper    5    66     1    135    0-01:52:15
00-17-0D-00-00-38-16-1A    7      Oper    2    15     1    117    0-01:51:56
00-17-0D-00-00-38-13-21    8      Oper    3    19     1     87    0-01:51:26
00-17-0D-00-00-38-14-39    9      Oper    3    24     1     29    0-01:50:28
```

3) Does it look like a mesh?

Check that all motes have two parents in the mesh using the `show mote` command on each mote in the network. In the example below, mote 2 has 8 neighbors, 2 of which are parents. There should be exactly one mote in the one hop ring that has only one parent. That is OK and expected. Calling the command as `show mote *` will print all motes in the network to the CLI.

```
> show mote 2
Mote #2, mac: 00-17-0D-00-00-3F-FC-04
   State:   Oper, Hops: 1.2, Uptime:    0-01:58:42, Age: 222
   Regular. Route/TplgRoute.
   Power Cost: Max 65534, FullTx 110, FullRx 65
   Capacity links: 200, neighbours 31
   Number of neighbors (parents, descendants): 8 (2, 6)
```

17.4 Does the Network Have the Building Blocks to BE GOOD?

This part of the network evaluation test involves answering three quantitative questions about the details of connectivity in the mesh. Those three questions are:

1) Are there enough motes in the one-hop ring?

All traffic in the network converges at the AP mote, the mote connected to the network manager. That single mote is critical for all data to be delivered. By extension, all the *one-hop* motes that communicate directly with the AP are important as well. The hardest working motes in the mesh will be in the one-hop ring. Those are the motes that are forwarding the most traffic from their descendants. The more one-hop motes, the better for a network as there is more opportunity to balance the traffic and to survive a single mote reset. We never want to build a system where removing one mote will cause the loss of many motes' data. As a rule of thumb, there should be at least 5 motes or 10% of the total, whichever is larger, in the one-hop ring. If you have a 120 mote network with 10 one hop motes, it is not guaranteed that it will fall apart. But you should have good quantifiable guidelines here so a non-expert can answer a yes/no question, with actionable instructions on what to do when the answer is no. In the example below, our AP (mote ID 1) has 6 neighbors. Since the AP by definition does not have any parents, all of these neighbors are children. In this example network there are 8 motes and 1 AP, so having 6 1-hop motes meets the 1-hop requirement.

```
Mote #1, mac: 00-17-0D-00-00-30-0A-F4
   State:   Oper, Hops: 0.0, Uptime:    0-02:12:37, Age: 0
   Power. Route/TplgRoute.
   Power Cost: Max 65535, FullTx 0, FullRx 0
   Capacity links: 250, neighbours 99
   Number of neighbors (parents, descendants): 6 (0, 8)
```

2) Does every mote have enough good neighbors?

This step involves waiting until 15 minutes after the last mote has joined, looking at all the discovered paths in the network, and making sure that every mote has enough good quality neighbors. The bare minimum is that every mote should have at least 3 good neighbors. A good neighbor is a neighbor that this mote can hear at >-75dBm or has >50% path stability. These paths do not have to be currently in use, they just have to be discovered and reported by the network. To check this detailed information, use the `show mote -a` command which also shows unused paths, and look at the list under the Neighbors heading. Count the number of rows where the link quality, or Q, is greater than 50%. In the example below, mote 9 has discovered all other 8 motes in the network and each one gets one line under the Neighbors heading. All values of Q listed are above 50%, so this mote has 8 good neighbors, meeting the requirement.

```
> show mote -a 9
Mote #9, mac: 00-17-0D-00-00-38-14-39
   State:   Oper, Hops: 1.6, Uptime:    0-02:19:09, Age: 400
   Regular. Route/TplgRoute.
   Power Cost: Max 65534, FullTx 110, FullRx 65
   Capacity links: 200, neighbours 31
   Number of neighbors (parents, descendants): 2 (2, 0)
   Bandwidth total / mote exist (requested):  864 / 864 (993)
       Links total / mote exist (requested):  7.0 / 7.0 (6.1)
       Link Utilization                    :  1.0
   Number of total TX links (exist / extra):  7 / 0
   Number of links     : 15
     Compressed        : 5
     Upstream   tx/rx  : 7 (7/0) (Rx10=0.0)
     Downstream rx     : 3
   Neighbors:
      -> # 1 Q: 84% RSSI: -56/0
      -> # 2 Q: 94% RSSI: -32/-35
      -- # 3 Q: 74% RSSI: 0/0
      -- # 4 Q: 74% RSSI: 0/0
      -- # 5 Q: 74% RSSI: 0/0
      -- # 6 Q: 94% RSSI: 0/0
      -- # 7 Q: 74% RSSI: 0/0
      -- # 8 Q: 74% RSSI: 0/0
```

3) Are any motes at or near their link limit?

In all current products, motes at 180 links or more indicate a risk of bandwidth issues in the network. To check this, look at the sm output again and focus on the Links column. In this example, the AP has 114 links and the busiest mote has 66 links, so all are safely below the link limit.

```
> sm
     MAC                  MoteId  State Nbrs Links Joins    Age StateTime
00-17-0D-00-00-30-0A-F4     1      Oper     6   114     1      0     0-01:53:26
00-17-0D-00-00-3F-FC-04     2      Oper     8    39     1    188     0-01:53:08
00-17-0D-00-00-38-16-2A     3      Oper     4    31     1    178     0-01:52:55
00-17-0D-00-00-3F-FC-18     4      Oper     3    58     1    167     0-01:52:47
00-17-0D-00-00-38-12-84     5      Oper     2    11     1    163     0-01:52:42
00-17-0D-00-00-3F-FD-3B     6      Oper     5    66     1    135     0-01:52:15
00-17-0D-00-00-38-16-1A     7      Oper     2    15     1    117     0-01:51:56
00-17-0D-00-00-38-13-21     8      Oper     3    19     1     87     0-01:51:26
00-17-0D-00-00-38-14-39     9      Oper     3    24     1     29     0-01:50:28
```

18 Application Note: Common Problems and Solutions

18.1 Introduction

Networks are built with the goal of providing reliable services while keeping power as low as possible on the wireless devices. Since links use energy, motes are given as few links as possible to adequately carry the expected traffic through the mote during the joining and steady-state phases of the network lifetime. The manager depends on the motes accurately reporting their service requirements and each path averaging better than 50% stability. If there is a bottleneck, meaning that any mote has run out of links due to power or memory constraints, there may not be enough bandwidth to carry all the traffic from the descendants of this mote.

Symptoms of a low-performing network are:

- Slow formation time
- Mote resets
- Large variation in packet latency
- Manager reports lost packets from any mote

The causes of these performance issues are typically one or more of:

- Poor connectivity - motes do not have enough neighbors with good quality paths
- Interference - in-band WiFi or Bluetooth is present or a strong out-of-band interferer is nearby to lower path stability
- Oversubscription - motes are reporting more than their accepted service requests allow causing congestion

Motes report their internal and path statistics in Health Report packets. These statistics are broken up into 2 or 3 packets and are generated every 15 minutes. In particular, check the reported path stability values on the paths that are currently being used by the mote. The mote reports the maximum and average size of its internal packet queue. Dust networks are provisioned so as to rarely have more than one packet at any mote at any time, so a nonzero average queue length usually indicates a problem.

The manager also keeps track of the network topology and the link assignments at every mote in the network. This perspective can immediately identify if any motes have run into link limits or have skipped a sequence number indicating a lost packet. Furthermore, the manager issues an alarm when a mote resets.

Interference can also be the result of many co-located Dust networks. In the current product line, networks are not synchronized in terms of time or bandwidth allocation so transmissions from one network can occur at the same time and on the same channel as another network. Measures have been taken to reduce the chance that this inter-network interference will cause serious performance issues, but it is possible to see overall path stability be lower in the a multiple co-located networks environment, and is a function of the total combined amount of traffic. Note that lower path stability may not necessarily translate into lower data reliability, severe interference will need to occur for that to happen.

18.2 No Motes Join

Reasons that no motes at all join to the manager include:

- The manager is not running.
- There is no AP mote connected to the manager.
- There is no antenna connected to the AP mote.
- The network ID and/or join key of the manager do not match the security credentials of the motes.
- The Access Control List on the manager does not include any of the motes.
- The motes are all placed too far away from the AP.
- The motes are not powered on.
- The sensor firmware on the motes is not sending the join command correctly.

18.3 A Collection of Motes Doesn't Join

If some motes join and others do not, you have at least established that the manager and AP are functional. Reasons that some motes won't join can include:

- Some motes are placed too far away.
- Max motes on the manager has been reached.
- Some of the motes do not have the correct security credentials to join this manager (network ID, join key, ACL entry).
- The motes that are within range have been configured as leaf nodes.

18.4 One Mote Doesn't Join

If the number of motes that fail to join is small relative to network size (i.e. 1 in 100), then potential reasons include:

- That mote has an RF problem (like it is broken or the antenna is not attached).
- That mote has the wrong security credentials.
- That mote is not powered up.
- Max motes has been reached.
- That mote is placed too far away from the rest of the network.

18.5 One Mote Gets Lost and Rejoins Over and Over

Motes should stay connected to the network indefinitely. Possible reasons for a single mote to join and drop off the network and join again include:

- A power supply problem on the mote is resetting it.
- The RF connectivity to neighbors is marginal.
- The RF connectivity to neighbors is highly transient and unstable.
- The mote is in a location where RF connectivity can be severed and then re-established (like in an enclosure or behind a large obstacle).

18.6 Devices Within Operating Range Have Bad Path Stability

It's probably interference, place them closer together to boost SNR.

18.7 I Need to Install a Repeater but I'm Already at Max Motes

Repeater needed for connectivity: Remove one mote and place the repeater, or rearrange motes to shorten paths.

Repeater needed for 1st hop bandwidth: Cut back on reporting rate, or move a mote from farther out into the 1st hop ring.

18.8 Data Latency is Higher than I Expect

Data latency can be lowered on an individual device at the expense of battery life (for the mote and its ancestors) by shortening the service period in a request but keeping publish rate unchanged. Latency can also be improved network wide by increasing the base bandwidth.

18.9 The Network is Using Paths that Don't Look Optimal

The network continually tries to optimize for lowest power - part of which includes trying new paths periodically. There are other considerations besides path stability that come into play.

19 Application Note: Changing Provisioning Factor to Increase Manager Throughput

19.1 Introduction

Managers guard for short-term changes in path stability through *provisioning*, assigning links to motes as a function of the traffic they generate. By default, the provisioning factor in a SmartMesh network is 3x - for every packet expected to pass through a mote, it gets three links. This allows a device to ride through temporary stability drops down to 33% without risk of its queue filling, which could result in lost data. However, there are a limited number of links available at the manager's access point, so provisioning places a cap on packet throughput.

For applications where manager throughput cap is limiting, breaking the network into smaller subnetworks is the preferred method for increasing total throughput. If this is not an acceptable solution, the customer can modify the provisioning factor to increase throughput, within limits. Dust recommends that provisioning never be set lower than 1.5x - path stability dips to < 70% are not uncommon in customer networks we've observed, and without clear knowledge that the network is operating in a low-noise, low-multipath environment, setting it lower is risky. In general, where paths are > 67%, set the provisioning factor to the reciprocal of the lowest observed path stability.

Note that SmartMesh managers use links for functions other than carrying data traffic, *e.g.* sending Keep Alives with enough retries to avoid a path alarm. Because of this, some motes may have more links to the access point than required by traffic alone. If the access point has reached its link limit, these links prevent any other motes from adding links for bandwidth purposes; it is more difficult to approach the limits discussed below in larger and/or deeper networks.

19.2 Changing Provisioning: IP VManager

Each AP in a VManager network supports 40 packet/s with 3x provisioning. Setting provisioning to 1.5x achieves a maximum throughput of 80 packets/s, but requires that all paths have > 70% stability.

To change the provisioning factor using manager CLI (here 1.5x):

```
$> su becareful
#> config seti BWMULT=150
#> reset network [--reload]
```

19.3 Changing Provisioning: Embedded IP Manager

Embedded IP managers with external RAM have 223 links dedicated to upstream data (150 links when no external RAM is used) on a randomized slotframe from 256-284 slots (270 average). Each slot is 7.25 ms. With 3x provisioning, this is 36.1 packets/s. Setting provisioning to 1.5x achieves a maximum throughput of 72.2 packets/s, but requires that all paths have > 70% stability.

To change the provisioning factor using manager CLI (here 1.5x):

```
> set config bwmult=150

> reset system
```

To change the provisioning factor programmatically, use the *setNetworkConfig<bwMult>* manager API.

19.4 Changing Provisioning: WirelessHART

WirelessHART managers have 737 links dedicated to upstream data on a 1024-slot superframe, where each slot is 10 ms. With 3x provisioning, this is 24.0 packets/s. Setting provisioning to 1.5x achieves a maximum throughput of 48.0 packets/s.

To change the provisioning factor (here 1.5x), you need to modify the *link_oversubscribe* parameter in the `dcc.ini` file in `/opt/dust-manager/conf/config/dcc.ini`.

```
# Oversubscribe coefficient for link (1.0 no oversubscribing)
# Range: 1-100
 link_oversubscribe = 1.5
```

✓ By default, the `dcc.ini` file does not exist. If you haven't already made parameter changes, you'll have to create the `dcc.ini` file to change the provisioning factor. Be sure to create it in the directory listed above.

20 Application Note: Debugging Congested Networks

20.1 Introduction

SmartMesh networks are designed to deliver every packet accepted by each mote from each sensor. If a packet is accepted by a mote but does not make it to the manager, this packet is classified as *lost* and counts against the *reliability* of the network. The reliability statistic that the manager reports is the ratio of non-lost packets to the total number of accepted packets.

There is another important metric called *availability.* Availability is the fraction of times that the sensor was able to hand its packet off to the mote when it wanted to. SmartMesh networks are generally provisioned with enough links to ensure 100% availability in addition to 100% reliability, but we do not directly measure availability because it is an application-layer metric. The only time that a mote is unable to accept a packet is when it has a full queue of packets, we call this a *congested* mote. A congested mote doesn't have enough upstream links to support its local and forwarding traffic. To determine if a network is in danger of losing availability, one must look for congested motes.

20.2 Respecting Services

SmartMesh managers use a *service* model to assign bandwidth in a network. In this model, each sensor application is responsible for figuring out how much data it needs to send and the associated mote is responsible for requesting enough bandwidth from the manager. In short, the responsibility of the application is to:

1. Calculate the packet generation interval for each separate data flow
2. Request a service for the sum total of all these data flows (these can be separate in SmartMesh WirelessHART but only one service in SmartMesh IP)
3. Wait for confirmation that this service, or a faster service, has been accepted by the manager before publishing
4. If no confirmation arrives, re-request the service after a timeout.

There are many details about the service model that are different between SmartMesh WirelessHART and SmartMesh IP. Refer to the respective guides for more information (SmartMesh WirelessHART Services and SmartMesh IP Services).

20.3 Estimating Availability

If you know that a mote is generating periodic data and you know the period, you can estimate how many packets should be received at the manager every 15 minutes. For example, if all motes are reporting once per second, you should expect 900 data packets per 15 minutes. On top of this, the mote sends three health report packets per 15 minutes and may also send responses to manager requests and path alarms. Availability typically is 100% or will drop considerably, so anything around 903 packets per 15 minute interval should be considered good in this example.

When there are fewer packets sent per interval, it gets more difficult to ascertain if they were all accepted by the mote and successfully received by the manager. Networks with less reporting have fewer links so the latency is generally longer which can push packet into the next 15 minute interval and a couple packets plus or minus can be a big fraction of the total number reported. In the following capture of mote statistics, all motes are reporting once per 5 seconds so we expect about 183 packets per interval. Mote 11 has a suspiciously low number in the PkArr column which tells us how many packets arrived during the interval. While the examples shown below show SmartMesh WirelessHART manager statistics, the same concepts apply to SmartMesh IP networks.

```
> show stat short 0
It is now .................. 06/06/12 16:30:17.

This interval started at ... 06/06/12 16:15:00.

 --------------------------------NETWORK STATS---------------------------------

PkArr  PkLost  PkTx(Fail/ Mic/ Seq)  PkRx  Relia.  Latency   Stability

 1806       0  5395(2430/   0/   0)  3098    100%   3.61s      54.96%

 ---------------------------------MOTE STATS-----------------------------------

 Id PkArr PkLst PkGen PkTer PkFwd PkDrp PkDup Late. Jn Hop avQ mxQ me ne Chg  T

  2   181     0   181     0   356     0    16  1.34  0   1   0   4  0  0   0 22

  3   182     0   182     0   194     0    20  1.49  0 1.2   0   3  0  0   0 22

  4   182     0   183     1    98     0    17  1.63  0   1   0   4  0  0   0 22

  5   182     0   182     0   329     0    18  1.46  0 1.4   0   4  0  0   0 22

  6   182     0   182     0    82     0    16   1.7  0 1.2   0   2  0  0   0 22

  7   154     0    --    --    --    --    42  20.7  0 2.6   -   -  -  -   -  -

  8   183     0   157     4    43     0    30  2.66  0 2.3   0   1  0  0   0 22

  9   183     0   182     0     0     0    30  3.39  0   2   0   2  0  0   0 22

 10   182     0   188     6    65     0    45  2.96  0 2.1   0   3  0  0   0 22

 11   166     0   169     1     0     0    42  17.2  0   3   1   4  0  0   0 22
```

Reliability here is 100% because there are no entries in the PkLst column, but the application was expecting more packets from motes 7 and 11 so availability < 100% for these motes. In general, you will not know how many packets the sensor is trying to send so you need to look instead for signs of congestion.

20.4 Identifying Congestion

In addition to the estimated availability, congestion can be identified by larger-than-expected latency or by looking at the mote queues in the stats. As a rule of thumb, motes that have an average queue length (avQ) greater than 0 are in danger of seeing congestion at some point. Motes that have a maximum queue length (mxQ) of 4 or more may have experienced acute congestion during the interval. Finally, congested motes typically have higher latency than their peers. Mote 11 in the above stats meets all of these criteria and was definitely congested during the previous 15 minutes.

A missing health report can also indicate congestion. In this scenario, when it was time for the mote to generate a health report, its queue was full and the mote was unable to complete the action. In the mote stats, this manifests like the mote 7 row above. We do not directly get a report on the queue occupancy, but we do still have the lower PkArr than its peers and high latency. When a mote does miss a health report, it continues to keep increasing its counters so that the next health report still summarizes the missing information.

When a mote is congested, it will start sending NACKs to its children when they try to unload their packets. This reduces the effective upstream bandwidth at the children of the congested mote and can, if there isn't enough provisioning margin, lead to the children becoming congested themselves. This process can repeat down through the mesh. Because of this, a mote three hops deeper than the problem mote may get congested and see that its sensor is backing off. This results in fewer data packets from this mote being received at the Manager than expected. In order to find the source of the congestion, the upstream route of each congested mote should be traced towards the AP. The lowest-hop mote that is congested is likely the source of the congestion of its descendants.

20.5 Bandwidth Model

The manager tries to give each mote three times as many links as it theoretically needs to carry local and forwarded traffic in a 100% stability network. This means that the network can operate down to 33% stability, though note that 33% stability often means that it is 100% for a little while and then 0% for a while later which is less conducive to successful multi-hop data collection than a constant two-failure-one-success pattern. If there is congestion anywhere, it means that something has broken down in this model, and one of the following is true for the source of congestion:

- It recently lost a parent due to a path alarm
- Path stability is below 33%
- It ran out of assignable links due to power or memory constraints
- A descendant is reporting more than allowed for its service/base bandwidth level

Path alarms can be monitored by subscribing to the manager notifications and are typically induce very temporary congestion that is resolved if the mote has a sufficient number of good neighbors. If the mote does not have enough good neighbors, the path stability could consistently be low and cause chronic congestion, and this can be due to interference. See the Identifying and Mitigating Interference App Note for guidance.

A `show mote` command can be issued on the manager CLI to see how close the congested mote is to its link limit. Specifically, you should check these two rows:

```
Neighbours: 27 (max 32). Links: 34 (max 100)
  Links per second: 4.882812 (limit: 11.648407)
```

The indicated number of links here, 34, is safely below the maximum of 100 and the power-induced limit of 11.6 link/s is safely above the current 4.9 link/s. If any mote is approaching the link limit, repeater motes should be added in close proximity to these motes.

Finally, the manager expects that each sensor will respect the service model. If a sensor is going to report at a faster rate than that specified in the base bandwidth for the network, it should request a faster service. Still, there is no guarantee that the sensor will do this properly so it should still be considered as a potential root cause.

To confirm that the mote is not overstepping its requested service level, the `show mote` result can again be used:

```
Bandwidth:
      output         planned   0.3906 current     0.3906
     global service            0.3614   delta    -0.0108
      local service  goal      0.0250 current     0.0250
    guaranteed for services    0.0250 for child   0.3241 Free   0.0415
```

Here we are looking at the local service line. The first value of 0.025 here is the pkt/s that the manager thinks the mote has requested including both the service and the base bandwidth. The current value is the same which indicates that the manager has indeed assigned this bandwidth. The negative value of delta indicates that this mote ostensibly has enough bandwidth to support all local and descendant traffic. At 0.025 pkt/s, the mote should generate at most 36 data packets in each 15 minute interval. The mote stats shown earlier, specifically the PkArr column, can be used to confirm that the mote is not generating more than this maximum level and exceeding its allotment.

20.6 Mitigating Congestion

If your SmartMesh network is congested at several locations, it may be that the path stability throughout the network is too low to function properly. In this scenario, the preferred response is to increase the mote density in the deployment. These additional motes, spread out among the previously deployed motes, can multiply the number of potential paths to choose from and thereby increase the overall path stability. If the new motes are not reporting additional data, this solution does not increase the total egress bandwidth required, and if anything, it should decrease the average power of the existing motes.

If the density of the deployment cannot be increased, we can increase the number of links at the congested motes. If the congestion is global, there are two equivalent ways of increasing the number of links at every mote in the network. Either the provisioning can be increased or the base bandwidth can be decreased. If the congestion is localized to one branch in the network, the motes in this branch can request faster services. When we increase the number of links, the motes with the link increases will have corresponding power increases. Furthermore, any of these increases requires extra receive links at the Access Point. If the Access Point does not have extra bandwidth available for allocation, it may be that the network has to be split into two networks to support the extra links.

21 Application Note: Identifying and Mitigating the Effects of Interference

21.1 Introduction

SmartMesh networks are designed to tolerate interference with minimal performance degradation. If we compare an environment with interference to one without interference, we expect small increases in average mote power and latency. However, in some cases interference can be strong enough to significantly impact performance, typically manifesting with the following symptoms:

- Mote resets
- A large number of 15-minute path stability values < 60 %, even at RSSI > -70 dBm
- Upstream latency > 2 seconds for motes less than 3 hops deep
- Average queue occupancy > 1, or max occupancy > 3
- Reliability < 99.9%, either as a consequence of motes resetting or high latency

Interference can take the form of an in-band interferer such as an 802.11g wireless router, or an extremely loud out-of-band interferer such as WiMax. Use of a spectrum analyzer (even an inexpensive WiFi sniffer such as a Wi-Spy) can help identify the region of the spectrum being jammed. Examples include:

- Fast-hopping interferers such as Bluetooth will appear as a uniformly raised noise floor
- 802.11 WiFi routers will appear as broad peaks across several of our channels
- Out-of-band interferers may not show up at all, but could still saturate receivers in-network

A directional antenna coupled with a spectrum analyzer may help pinpoint the source of the interference, but it is often possible to deduce the presence of an interferer without using a spectrum analyzer at all - network statistics can often be used to infer the presence of an interferer, and additionally determine if the interferer is significant enough of a problem to warrant mitigation.

21.2 Checking RSSI and Path Stability

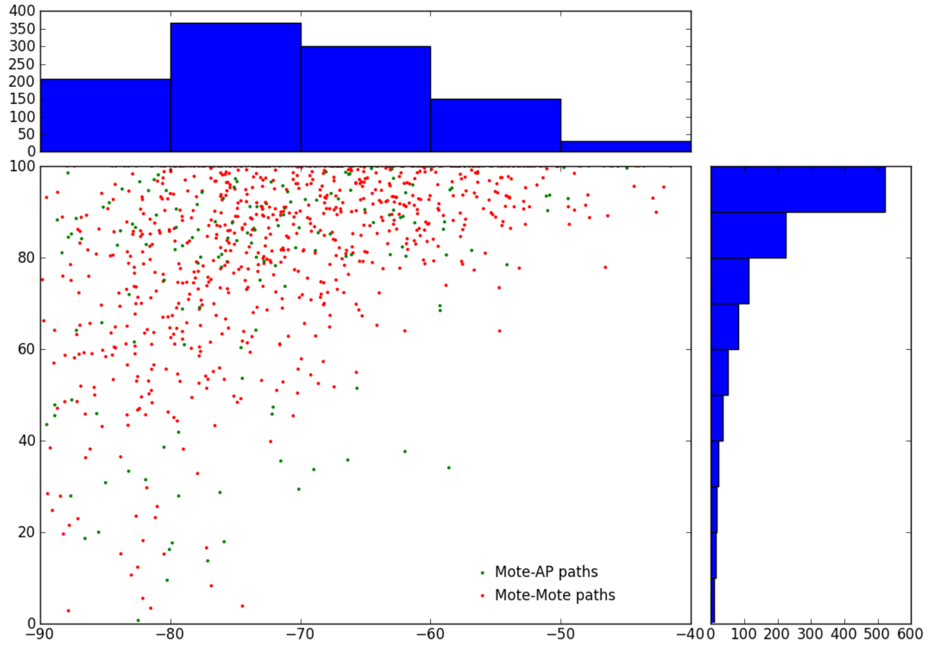
We only really care about interference if it has an impact on the path stability in the network. To see if this is the case, we recommend capturing several hours of Health Report packets from the network. An easy way to do this is by using the SmartMesh SDK `HRListener.py` tool. With this information, a waterfall curve can be plotted. For each upstream path in each Neighbor Health Report, plot a single point on the waterfall with the reported RSSI on the x-axis and the average 15-minute path stability on the y-axis. For example, a mote with two parents would contribute one Neighbor Health Report every 15 minutes, and would contribute two dots to the plot every 15 minutes.

Here is a snippet from a `receivedHRs.log` file which is generated by the tool:

```
2014-07-09 11:52:08,233 [INFO] from 00-17-0d-00-00-3f-fc-04:
- Neighbors:
    - neighbors:
        -item 0
            - neighborFlag        : 0
            - neighborId          : 10
            - numRxPackets        : 45
            - numTxFailures       : 0
            - numTxPackets        : 0
            - rssi                : -58
        -item 1
            - neighborFlag        : 0
            - neighborId          : 1
            - numRxPackets        : 5
            - numTxFailures       : 545
            - numTxPackets        : 1275
            - rssi                : -72
```

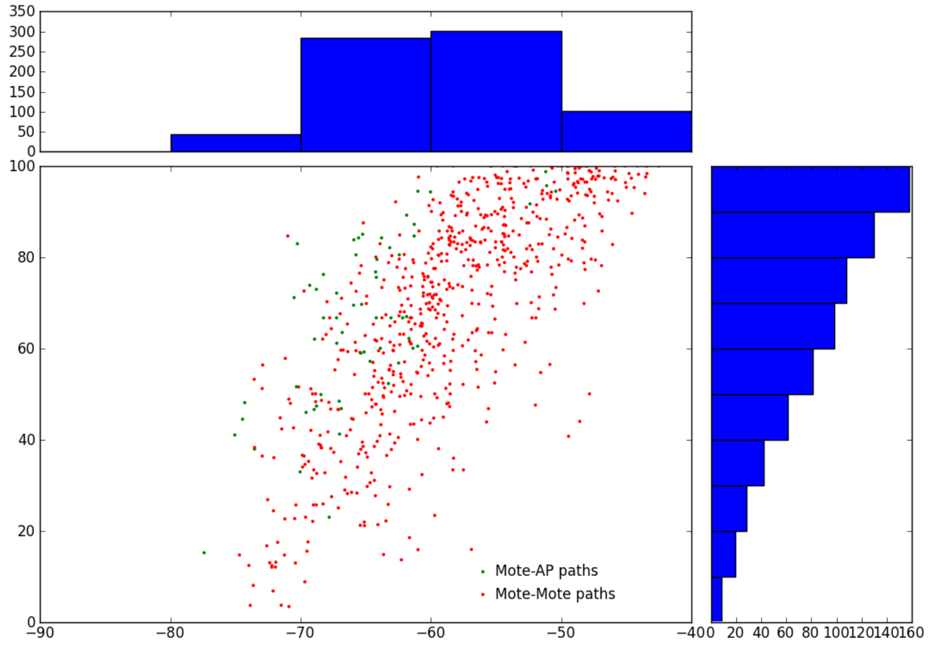
We are looking for all paths with `numTxPackets`>15, so we don't look at "item 0", the path to `neighborId`=10. We do, however, look at the path to `neighborId`=1. The stability for this path is 1-545/1275=57%. We would plot a point at (-72 dBm, 57%) on the waterfall. In large networks, the waterfall curve can get crowded so we recommend also plotting horizontal and vertical histograms to easily see the density.

A good waterfall curve looks like this:



In the top histogram, verify that there are a number of paths used in the -90 to -80 dBm range and also in the -60 to -50 dBm bin. In the right histogram, verify that a significant fraction of the paths are 80% or better and that there aren't many paths below 50%. In the scatter plot, we expect most paths better than -70 dBm to be 60% or higher stability. By breaking out mote-mote paths and mote-AP paths, hardware or interference differences specific to the AP can be seen.

Contrast the good waterfall with this one:



The whole curve here has shifted right and we see very few paths below -70 dBm. This type of waterfall is exemplary of either bad receiver or interference because devices are having a hard time hearing weak signals. There is still a chance that this network could meet customer performance expectations, but special care should be taken to ensure that all motes have sufficient connectivity in a deployment like this.

During development, the developer should deploy a test network with their specific hardware in a "known good" environment similar to their target deployment environments and without measurable interference. A waterfall curve generated from this test deployment should be used as the best case against which any in-field data can be compared. Antenna choice in particular can have a large impact on receive sensitivity, so it is not possible for Dust to provide a single Gold Standard waterfall curve to which all other designs should be compared.

21.3 Checking Mote Latency, Queue Lengths and Reliability

Some statistics can be checked using the `show stat` command on the manager CLI. Relevant here are the Network Reliability and individual Mote Latency values. In the example below, the reliability is perfect and the latency values (reported in milliseconds) are all well below the 2 second upper bound we discussed earlier.

```
> show stat
Manager Statistics --------------------------------
   established connections: 1
   dropped connections    : 0
   transmit OK            : 1687
   transmit error         : 0
   transmit repeat        : 1
   receive  OK            : 2028
   receive  error         : 33
   acknowledge delay avrg : 34 msec
   acknowledge delay max  : 112 msec
Network Statistics --------------------------------
   reliability:  100% (Arrived/Lost:   3677/0)
   stability:     72% (Transmit/Fails: 102701/28964)
   latency:     1300 msec
Motes Statistics ----------------------------------
   Mote    Received   Lost  Reliability Latency Hops
   #2           113      0   100%        240    1.4
   #3            95      0   100%        230    1.6
   #4            76      0   100%        170    1.3
   #5            51      0   100%        120    1.1
   #6            57      0   100%        160    1.1
   #7            97      0   100%        440    2.3
   #8            48      0   100%        140    1.0
```

The Device Health Report, also captured by `HRListener.py`, provides information related to the queueing and congestion levels at a mote. Here is an example:

```
2014-07-09 11:52:42,542 [INFO] from 00-17-0d-00-00-3f-fd-57:
- Device:
    - badLinkFrameId      : 0
    - badLinkOffset       : 0
    - badLinkSlot         : 0
    - batteryVoltage      : 3160
    - charge              : 88
    - numMacDropped       : 0
    - numRxLost           : 0
    - numRxOk             : 2
    - numTxBad            : 0
    - numTxFail           : 0
    - numTxOk             : 35
    - queueOcc            : 33
    - temperature         : 25
```

First we consider the `queueOcc` value, reported as an integer. The leading 4 bits of the integer is the maximum queue length experienced in the past 15 minutes, and the final 4 bits is the mean queue length, also over the past 15 minutes. So the value here of 33 = `00100001` means that the maximum queue was 2 and the mean queue was 1. According to the rules presented at the top of this document, the mean queue occupancy value indicates that the mote is congested. Second, we can look at the value of `numTxFail` which counts the number of times the mote refused to accept a locally generated packet because of a full queue. If this is ever non-zero, as it is here, it is also a sign of congestion.

21.4 Mitigation

There are several mitigation strategies for dealing with interference once it has been identified. The correct strategy depends on the temporal character and strength of the interference and on the performance goals for the network. The following sections detail each strategy. Note that most strategies that change network behavior require a network reset to take effect.

21.4.1 Remove Source of Interference

This requires the use of a directional antenna to locate the source, or knowledge of what other devices might be transmitting in vicinity. Often the interferer is legally permitted and it can't be moved. However, an IT department may be amenable to running the multi-band WiFi outside the 2.4 GHz range. In another case, a third party WiMax interferer was found to be transmitting above the allowed limit and legal proceedings were successfully undertaken to get it turned down.

21.4.2 Blacklisting

If the interference occupies a small fraction of the spectrum, it can be blacklisted around. SmartMesh IP systems can be blacklisted down to as few as 7 of the 15 available channels. See the "Channel Blacklisting" section of the SmartMesh IP User's Guide for details and caveats.

21.4.3 Add Repeater Motes

The manager will try to work around paths that are most severely affected by interference, but there may be motes that don't have acceptable alternative parents. For these motes, a repeater device can be added between the troubled mote and its current parents. By halving the distance the signal needs to travel, the SNR should increase enough to raise the stability seen by the child mote. While we have added another device to the network, on average the path stability for existing devices will increase so their average current will decrease. When following this strategy, repeaters should be filled in starting at the manager and working outwards.

21.4.4 Increasing Parent Number

Networks with low stability due to interference may have mote resets due to the manager having difficulty maintaining reliable downstream communication with all the motes. Downstream transport eventually fails over to broadcast flooding, so the more parents each mote has in the network, the higher the chance than any given mote will receive a downstream packet. The parent number can be changed network-wide using the *setNetworkConfig()* API with the `numParents` parameter. More parents means that the mote schedule gets busier and each mote's average current will increase. For LTC5800 parts, busy routers will be the least impacted by going from the default two parents to three parents, with an average current increase around 15%. Low-data leaf motes could see their average current increase by up to 32%. In cases of extreme interference, the number of parents can be increased to four, with another similar increase to average current expected.

21.4.5 Increasing Downstream Retries and Timeouts

If identifying truly lost motes is not a priority, there is another way to increase downstream reliability without significantly increasing power: increase the maximum number of retries the manager will send before declaring a mote lost. By default, the manager will try each packet a maximum of 5 times. We can double this limit using the manager CLI and the following commands:

```
su becareful
seti ini numretr 10
```

These changes alone should make a shallow network of under about five hops more resilient. If the affected network is deeper than this, it could be that the downstream packets are timing out before reaching their destination. Increasing the retry timeout on the manager spaces out the time between subsequent attempts at reaching a mote, again with the effect of pushing out the time to identify a lost mote. In these scenarios, we recommend building the network according to "Application Note: Building Deep IP Networks".

It is difficult to specify exactly how the reset rate is reduced by these changes as reset behavior is dominated by the worst paths in the network at the worst times of interference, two limits that are hard to predict. In certain high-interference deployments, however, we have seen these changes reduce resets from about 10 per day to fewer than 1 per day.

21.4.6 Increasing the Provisioning Factor

If interference is causing latency increases but not causing mote resets, giving the motes more upstream links can help. More upstream links reduces congestion and can help overcome low path stability, primarily in multi-hop networks. The default value for provisioning is 300%, meaning that on average, motes have at least three transmit attempts for every packet they need to send. This can be increased to a more conservative value of 400% using the *setNetworkConfig()* API with the `bwMult` parameter.

Going from 300% to 400% with LTC5800 devices, leaf motes should not experience significant increases in average current; routers can have average current increases of around 5%.

In extreme conditions, the provisioning can be raised up to 600% and router current would then increase by around 15%.

21.4.7 Increasing the RSSI Floor on New Paths

The manager, by default, will want to add links to test out newly discovered paths with RSSI measured above -80 dBm. If waterfall data indicates that very few paths below -70 dBm have decent stability, we can increase this new path threshold accordingly with the INI setting:

```
su becareful
seti ini rssith -70
```

The trade-off here is that if we are too conservative, the network will be built with more hops than necessary and will have higher average current and latency.

21.4.8 Adding a Narrowband Filter

If the interference is coming from outside the 2.4-2.48 GHz band, a narrowband (e.g. BAW) filter can be added to reduce the received interference. For this solution, a few guidelines should be followed:

- Select a filter with near-constant group delay for minimal impact on inter-symbol interference
- Typically the filter will introduce a 2 dB insertion loss which affects both receive and transmit paths, so overall paths will lose 4 dB of link budget
- Ensure proper grounding as specified by the filter manufacturer
- Choose passband appropriately over the full operating range (temperature, voltage, etc.) for the device; your goal is to filter the expected interference without degrading the performance of the upper and lower channels

22 Application Note: Obtaining Accurate Timestamps

22.1 Time

All devices in a Dust network share a sense of time. This allows a sensor application to use network time to provide for accurate timestamping of sensor measurements. Accuracy can be as tight as 10's of μs under ideal conditions. The process is similar across families:

1. The sensor processor takes a measurement and strobes the mote's time pin
2. The mote returns its local time (in UTC and ASN)
3. The sensor processor places a timestamp and data in a packet
4. Mote forwards packet to manager for delivery to the host application
5. Host post-processes the data notification as needed to account for differences between network time and "absolute" time

Timestamp accuracy of a measurement is determined by various uncertainties in the system, which will be discussed in this application note.

22.2 References

- RFC 5905 defines version 4 of the Network Time Protocol (NTP)
- IEEE1588-2008 defines version 2 of the Precision Time Protocol (PTP)
- IEEE C37.238 is a spec for using 1588 in power systems. It specifies using Ethernet and various settings for synchronizing widely separated installations.

22.3 VManager IP Systems

It is recommended to connect Access Points to a GPS clock when seeking accurate timestamps in a VManager network. When this is done, the ASN will track UTC time to within 10 μs, which means that this resolution is available everywhere in the network including at the motes. Using this system obviates the need to account for the timing differences described in this document.

22.4 Embedded Manager IP Systems

A flowchart of the procedure for using network time is shown in Figure 1. The SmartMesh IP manager starts "UTC time" at 20:00:00 UTC July 2, 2002 when it starts up. (The manager's *setTime* API has been deprecated - it isn't possible to set the time prior to network formation). From that point on, the manager will slowly drift away from "absolute" time, resulting in **Manager UTC uncertainty**, as there is currently no mechanism to continually correct time (e.g. by NTP as is available in the SmartMesh WirelessHART manager). By using the manager's time pin, a host application can correct for this drift. It is not necessary to set a valid UTC time on the manager to make accurate timestamps, since time is returned in both ASN and UTC on the mote.

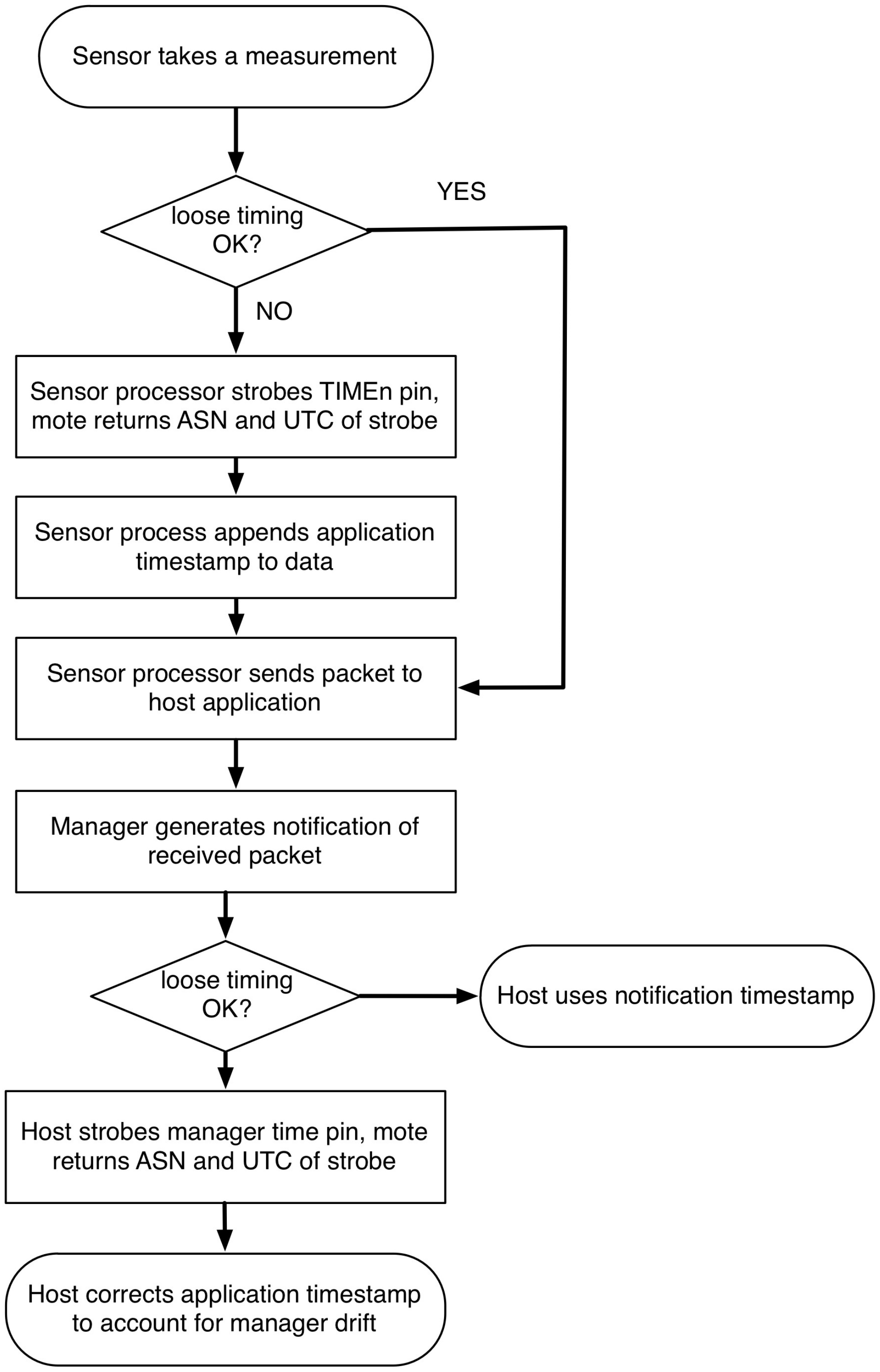

Sensor takes a measurement
loose timing OK?
YES
NO
Sensor processor strobes TIMEn pin, mote returns ASN and UTC of strobe
Sensor process appends application timestamp to data
Sensor processor sends packet to host application
Manager generates notification of received packet
loose timing OK?
Host uses notification timestamp
Host strobes manager time pin, mote returns ASN and UTC of strobe
Host corrects application timestamp to account for manager drift

Figure 1 – Timestamping in an Eterna LTC5800/5901/5902 manager network

22.5 Loose Timing

If only loose (100's of ms) timing is required, no application timestamp is required - the host application can use the timestamp from the *data* or *ipData* notification. This timestamp has **queuing uncertainty**, which can be 10's of ms. This timestamp still has Manager UTC uncertainty, which we will discuss correcting in the context of timestamped data.

22.6 Tight Timing

If sub-millisecond timing is required, the sensor processor can take a measurement and strobe the mote's time pin to get a network timestamp to include in the packet. The mote returns its local time (in UTC and ASN), which has two uncertainties - the **mote local time uncertainty**, which is a function of the path stability, temperature variation among motes, topology, etc., and the mote's **time pin uncertainty**, which is a datasheet property for the HW family. Either ASN or UTC timestamp can be included in the packet - using ASN makes it clearer that the UTC time may not in fact be accurate.

When the Application Processor receives a data notification, it must trigger the manager's time pin, thus getting the current ASN or UTC time to within the manager's **time pin uncertainty**. This allows the application to use the timestamp in the packet, along with the current timestamp from the manager to eliminate **Manager UTC uncertainty** and calculate the absolute timestamp of the packet to within the accuracy of time at the host.

22.7 Highest Precision Timing

To get highest precision timing, all sources of uncertainty must be accounted for, and minimized if possible:

- Transit time (latency) adds **transit uncertainty**, which is a function of network topology and activity. If the packet was generated on a very low activity network, it could take 30 s to arrive at the manager – if the manager drift is 50 ppm, manager drift during transit would add 1.5 ms of uncertainty to the calculation. Transit uncertainty can be removed if the application processor periodically strobes the manager's time pin and measures manager drift – then it can calculate a running UTC offset to remove transit time induced uncertainty in addition to Manager UTC uncertainty. Under typical conditions however, transit uncertainty is a few μs.
- Any delay or uncertainty within the sensor processor related to taking a sample and strobing the mote's time pin are up to the integrator to determine and account for.
- A fast temperature ramp can dominate the **mote local time uncertainty**. This error will propagate to all descendants of the ramping mote. In cases where this effect can raise the error beyond the system requirement, the application can measure temperature at all motes and report alarms when significant ramps are detected. The host application can then choose to ignore data timestamped during these alarm periods or assign them more uncertainty than measurements taken during times where all motes sit at constant temperatures. Flat topology networks operating at a stable temperature will have the lowest mote local time uncertainty.

22.8 Quantifying IP Uncertainty

The following are values for the uncertainties in an IP network:

- **Time pin uncertainty** – +/- 1 µs worst case for an Eterna-based manager or mote.
- **Mote local time uncertainty** – for a mote at *h* hops, we expect the typical uncertainty to be $0.6h^{1/2}$ µs and a worst case of 30*h* µs. On top of this, temperature ramping up to our specified limit of 8 °C/min increases each distribution by 1.5 µs·min/°C.
- **Manager UTC uncertainty** – +/- 50 µs/s of up time worst case, +/- 2 µs/s of up time typical if the crystal has been characterized.
- **Transit uncertainty** – +/- 50 µs/s of packet transit time worst case, +/- 2 µs/s of transit time typical if the crystal has been characterized.
- **Queuing uncertainty** – < 50 ms worst case, assuming the packet is acknowledged by the mote. Only relevant if using network layer timestamps.

22.9 WirelessHART (Linux SBC-Based) Systems

The Linux SBC based (PM2511/DN2511/LTC5903) manager supports NTP for keeping the manager synchronized to an absolute time reference. The manager is connected to an NTP server to get UTC (global) time. The accuracy of the manager's notion of global time is characterized by the **Manager UTC uncertainty.** Connecting the NTP server with a dedicated connection could eliminate IP network contributions to uncertainty.

Network time (ASN) is kept separately by the Access Point (AP). The manager measures the AP's time by periodically strobing the AP's time pin, and comparing the resulting time with its own estimate. This measurement contains the AP's **time pin uncertainty**, which is a function of which AP is used (DN2510 or Eterna). When the AP's time differs by > 100 ms (configurable via the *max_utc_drift* ini parameter), the manager pushes a new UTC time mapping to all motes via unreliable broadcast. Reducing *max_utc_drift* results in more messages downstream – for example an LTC5800-based AP with 50 ppm drift would require a DS message approximately every 30 minutes to be within 100 ms of the manager's UTC time. To keep it to 10 ms requires a message every 3 minutes.

If the sensor processor is using UTC timestamps on the mote, then the **UTC mapping uncertainty** must be added to the UTC uncertainty. It is possible for a mote to miss several UTC updates, and thus be off >200 ms from the AP. In theory, using ASN timestamps could eliminate the UTC mapping uncertainty, but the WirelessHART manager does not provide a way to get a precise ASN – the *getTime* manager API returns the last ASN/UTC mapping that the manager measured, and so could be up to *max_utc_drift* ms in error.

Since many of the sources of uncertainty cannot be controlled or accounted for in a current SmartMesh WirelessHART manager, only loose timing (100's of ms) is available.

22.10 Quantifying WirelessHART Uncertainty

The following are values for the uncertainties in a WirelessHART network:

- **Manager UTC uncertainty** – this is a function of the manager, the NTP server, and IP network latency. It is not a function the AP. This has been measured to be ~1 ms on low-traffic Ethernet networks, but can rise to many 10's of ms when the Ethernet network becomes busy.
- **Time pin uncertainty** – this is expected to be -60/+600 µs for a DN2510 based AP or mote, and +/-1 µs for an Eterna-based mote.
- **UTC mapping uncertainty** - It is possible for a mote to miss several UTC updates, and thus be off up to +/- 300 ms from the AP.
- **Mote local time uncertainty** – With a DN2510-based AP, we expect the mean to be around $45h^{1/2}$ µs and a worst case of $150h$ µs for a mote at h hops.. With an LTC5800-based AP, we expect the mean to be $1.2h^{1/2}$ µs and a worst case of $30h$ µs. On top of this, temperature ramping up to our spec limit of 8 °C/min increases each distribution by 3 µs·min/°C.

22.11 Synchronous Events

Synchronous measurements or events are also possible with all products, with a slightly different procedure:

- The application uses the manager *getTime* API call to get current ASN.
- The host application broadcasts* an application-layer packet containing an event time (in the future). The message is sent 3 times to ensure delivery**. The future ASN can be used by all the motes to synchronize an event. A large enough future ASN should be chosen to ensure that every mote receives the future ASN and has time to prepare for the event - any value greater than 120 seconds is safe.
- Motes receive the packet and send a notification to their sensor processor containing the application-layer packet.
- Each sensor processor uses the mote time pin to determine the current time.
- Each sensor processor determines when it is time for the event and takes a measurement, or actuates, as appropriate. Assuming that the sensor processor is running an uncompensated 32 kHz xtal as a time reference (+/- 150 PPM), device to device variation in the absolute sample time on a 120 second timer could be off by +/-15 ms. Note that this requires a 24 bit timer. If this level of synchronization is unacceptable, the sensor processor can periodically poll the mote for the current ASN and adjust its timer accordingly. Conversion between ASN delta and time is:

delta time = delta ASN * slot length

where slot length for SmartMesh WirelessHART is 10 ms, and for SmartMesh IP is 7.25 ms.

- Each sensor processor can optionally execute the timestamp logic above for any data generated by the event.

*The destination (IP mesh layer or WirelessHART net layer) address is 0xFFFF**3 retries is chosen for unreliable downstream transport as a tradeoff between wait time and likelihood of all the motes receiving the command. More retries increases this likelihood at the cost of a longer wait (i.e. the future ASN must be farther out). Selective retries (to reduce duplicates) are possible if application level acknowledgements are provided by the sensor processor. The sensor process must discard duplicate commands (containing the same future ASN).

23 Application Note: Using Multiple Managers to Build Large Networks

23.1 Large Deployments

There are installations where there will be more sense points than can be supported by a single manager - we handle this today by placing multiple managers to cover the installation. Some or all of these managers and their motes are expected to have overlapping radio coverage. When possible, it is good practice to separate "co-located" managers by a few meters instead of placing them right next to each other - this is to avoid radio interference between their Access Points. Installations with multiple co-located managers will function perfectly well provided the total traffic in the vicinity is kept at a safe level.

Each manager controls its own network, and there are two critical ways that motes are separated into various networks. First, each network has a Network ID. This value is in every advertisement sent by the network and is used to filter packets once motes have joined. Second, the manager can have an Access Control List (ACL) which specifies the MAC address and join key of each mote that is allowed to join the network. By using Network ID and ACL together in different ways, motes in a large deployment can be partitioned into smaller networks. Controlling Network ID puts the onus on the mote to decide which network it is going to join. Controlling the ACL allows the manager to make the decision, but this can be bad for a mote trying to join a network for which it is not on the ACL.

Depending on the software license used, the SmartMesh WirelessHART manager can support up to 250 or 500 motes and the SmartMesh IP Embedded Manager can support up to 32 or 100 motes if external SRAM is used. Obviously, a deployment requiring more motes than this requires multiple managers, but multiple managers can also be deployed to increase the overall egress bandwidth of the deployment. Each SmartMesh manager supports 20-36 packet/s of egress bandwidth, so for example, a deployment with 100 motes reporting data once per second could be safely done with five SmartMesh managers.

VManager-based SmartMesh IP networks can support thousands of motes. Using VManager for large deployments is the recommended solution; this document details the solution for when a WirelessHART network is required or when the SmartMesh Embedded Manager is being used.

23.2 RF Limitations

For SmartMesh WirelessHART, there are 15 channels and 100 slots per second. This is enough *cell space* for an absolute maximum of 1500 packet/s to be transmitted on different channels and/or at different times. When multiple managers are sharing the same radio space, there will be collisions if traffic overlaps because the managers do not share their schedules or their precise sense of time with each other. Empirically, we find that it is safe to use about 25% of the cell space when co-locating managers, so in this case it is safe to add managers up to the point of 375 packet/s total egress bandwidth. As an example, suppose 1800 motes are deployed using four SmartMesh WirelessHART managers and each mote is configured to report temperature every 30 seconds. The total egress requirement here is 1800/30 = 60 packet/s so this deployment is well under the RF limit.

In SmartMesh IP, there are the same 15 channels but there are 138 slots per second. Running through the same calculations, the safe limit for co-located managers is 517 packet/s.

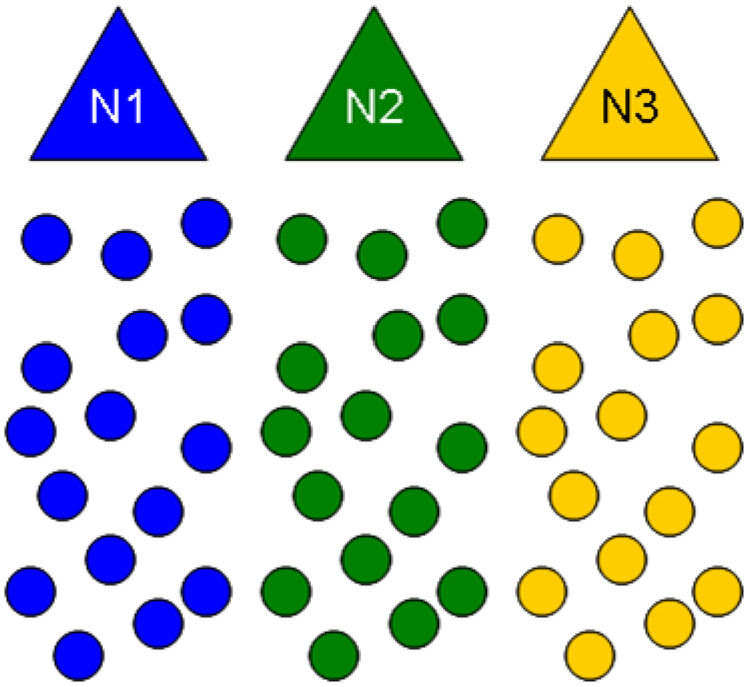
As traffic is added to overlapping networks, inter-network collisions will manifest as small decreases in path stability. SmartMesh managers have mitigation strategies to spread out the collisions so that they tend to affect all paths equally instead of completely killing any one path. Because Dust networks are low-duty cycle and low-power, interference from other Dust networks is less damaging than other forms of interference like WiFi.

23.3 Ideal Deployment Guidelines

In the ideal case, motes are divided equally among the available managers. If there are 48 motes and 3 SmartMesh managers, for example, then each manager is given 16 motes. To accomplish this, there are several steps:

- Three unique Network IDs are chosen and one given to each manager
- Sixteen motes are designated for each manager and programmed with the appropriate Network ID
- Each manager is programmed with an ACL containing its 16 mote MAC addresses and join keys
- In the field, the motes are placed so that each 16-mote network follows our deployment guidelines

In the figure below, these three networks partitioned by Network ID are represented by different colors. The motes are physically placed in locations such that the each colored network is well connected independent of any other network's motes.



Motes filter advertisements by Network ID, so a joining mote will only try to send a join request to a parent with the same Network ID as the joining mote. If a mote has the correct Network ID but the manager does not have that mote in its ACL (a mistake in this scenario), the mote will try to join but the join request packet will be dropped by the manager. In this case, the mote takes several minutes before exhausting its retries and resetting.

While this approach is the most secure and speeds up network formation, the optimal performance comes at the cost of configuration time. Each mote must be programmed to the correct Network ID, and installed in the right area on the site. Each manager must be populated with the correct ACL. If a replacement device or a repeater is needed in one of the networks, the replacement mote must be programmed to the correct Network ID and the correct manager must be given an ACL entry before the new device can join. Furthermore, if a mote moves from one location to another in the deployment space, it may leave its current Network ID and not be able to rejoin using a different Network ID for its new location.

23.4 Mote Behavior - Using Search

The mote application can use the *search* API prior to joining to scan for different Network IDs in the vicinity. In this mode, the mote listens to **all** advertisements regardless of the Network ID. For each advertisement it hears, the mote reports the Network ID of the sender, the signal strength, and the depth of the sender in hops. This gives the application enough data to intelligently select which network it wants to join. The application then sets the Network ID of the mote and issues the *join* command which tells the mote to start listening to advertisements for its particular Network ID. The mote then tries to join only this prescribed Network ID. We recommend programming the application with a preferred Network ID that the mote should try when first booting up. If no advertisements from this Network ID are heard after a timeout, the application can set the mote to *search* mode to gather the information to join an alternative network. This process also allows some mobility; motes that move and reset with neighbors having a different Network ID can discover this new Network ID and rejoin a different network in the multi-manager deployment.

23.5 Manager Behavior - Different Network ID and Shared ACL

In order to use the Embedded SmartMesh IP Manager with thousands of ACL entries, external SRAM must be installed and the manager must be version 1.4.1 or later.

SmartMesh managers can store thousands of mote MAC IDs and join keys in their ACL. If a multi-manager deployment is being planned, say with 1000 motes and 4 managers, each manager can be given the full 1000 motes worth of ACL and join keys, and its own separate Network ID. The mote applications can then use the *search* API to scan advertisements from all networks in the area as described in the previous section.

Alternatively, a multiple-manager deployment can be done with the same Network ID and with different ACLs. In this case, each manager must know which motes it is allowed to accept. The only justification for using this method is in the case where it is desirable to have motes join predefined managers. For example, if 300 motes are being split evenly among 6 SmartMesh Wireless HART managers, each manager could be given a different 50-mote ACL that would determine which motes end up being controlled by which manager. The downside of using this strategy alone is that a mote trying to join the wrong manager will not know that it is being denied. This mote will send in a join request packet through the parent that it heard advertising, and it will get a proper link-layer ACK from that parent. The joining mote will maintain synch with the network while waiting for a response from the manager, but the manager will drop the join request because the joining mote is not on the ACL. It takes a long timeout period before the mote will reset and try joining again. There are cases, however, where it is not easy to set Network ID for motes once they have left the factory and ACL separation is the only way to achieve balanced networks because all the networks must run using the same Network ID.

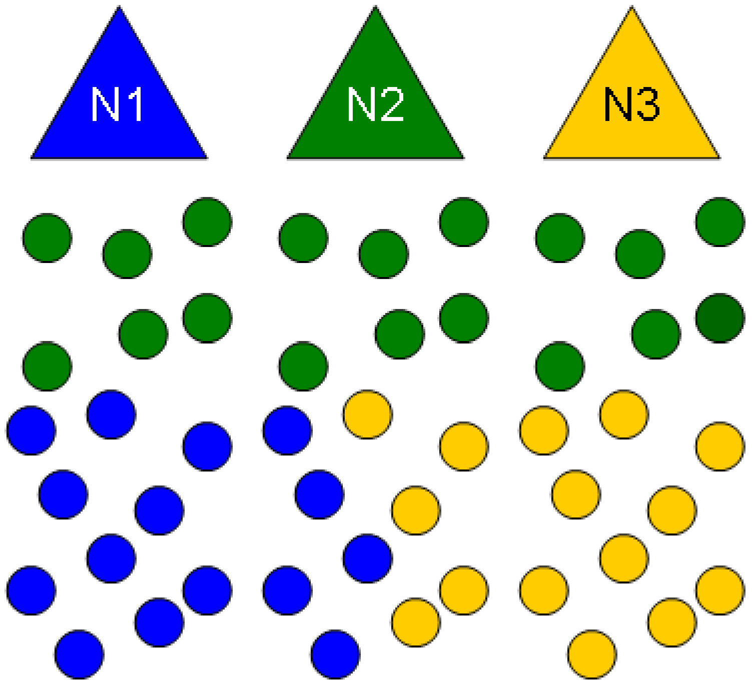
Providing that each network follows the deployment guidelines for range and connectivity, using ACL separation alone will eventually result in all motes ending up in their prescribed network, it may just take awhile. If a mote moves to a new area of the deployment outside of its original network, it cannot rejoin unless the manager of the new area is programmed with a new ACL entry. To enable mobility for a small number of motes, all managers in a deployment can have an ACL entry for these mobile motes. A mobile device leaving a network will then reset and rejoin a new network after hearing new advertisements.

23.6 Multiple-Manager Deployment Risks

There are a couple risks that should be evaluated when deciding which approach to take.

It is also possible to set all motes and managers to a single Network ID and not separate out the ACLs at all. If a single Network ID is used, motes will greedily join whichever network they happen to hear. This can result in motes not being assigned equally to the available managers as the manager that starts advertising first tends to start an avalanche effect of joining. This popular manager could reach its mote or bandwidth maximum having joined all the motes close to a clump of multiple managers and this could strand more distant motes. If this approach is taken, the application must be able to recognize that managers are filling up and reset motes intelligently in hopes that they will join under-used managers.

Before leaving the factory for deployment, the individual networks could be partitioned properly using separate Network IDs and ACLs and put into separate boxes for network 1, network 2, etc. But care must be taken during deployment to locate the motes properly as well. The individual deploying the motes might put all the motes from the network 1 box near the managers, stranding motes for network 2 beyond the range of the manager for that network. Even if some attempt is made at spreading the network out, care must be taken so that the connectivity deployment guidelines are followed for each network individually, not just for the deployment as a whole. Special attention should be paid to bottlenecks - there may be many motes around a mote from network 1 but they may all be from other networks. In this case, a follow-up visit is sometimes required to install repeaters, and these repeaters must be properly programmed with the specific Network ID.



Improper separation of the networks can lead to bottlenecks or starvation as shown in the above figure. Here the green manager has taken all the motes close to the manager and the yellow and blue managers are not close enough to their motes to allow them to join.

23.7 Setting Network ID and Join Key Over the Air

From the manager, the application can set the Network ID and join key on the motes over the air. Some customers use a fixed Network ID and join key during a test build at the factory and then reprogram the settings to unique values prior to packaging up the network for deployment. This is particularly useful for customers with sealed mote packages that cannot be physically connected for reprogramming. The new values take effect after a mote/manager reset.

24 Application Note: Using the SmartMesh Power and Performance Estimator

24.1 Introduction

This document describes some of the ways you can use the SmartMesh Power and Performance Estimator to make high level conclusions about how SmartMesh Networks operate. Many of these conclusions are not necessarily intuitive, but once you understand them, you will be much better equipped to make system-level decisions on how to plan out wireless sensor networks based on SmartMesh products. The analyses below use the SmartMesh IP platform, but the techniques hold equally true for SmartMesh WirelessHART networks, which are covered with the same spreadsheet.

The quantitative results described in this document were obtained using an older version of the SmartMesh Power and Performance Estimator, so expect slightly different currents and latencies when using an updated version. However, the qualitative results are what is important in this document and they are still valid.

24.2 Problem: What is my Battery Life?

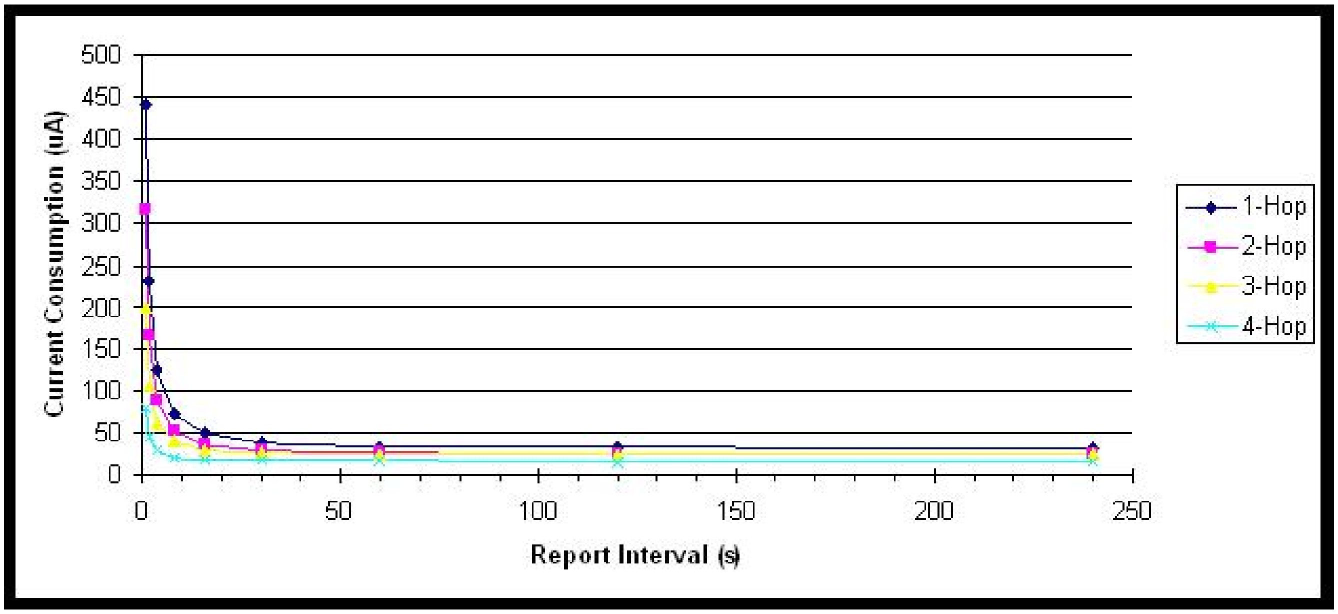
In the wireless sensor network WSN marketplace, battery life or power consumption is a key differentiating feature. SmartMesh products are uniquely well positioned to win in any battery life comparisons because they use the lowest-power radios and the best protocols to deeply duty cycle those radios. That said, it is a fact that at the moment one installs a device, they do not know what the battery life of that device is going to be. Will it be a leaf node in the mesh, responsible for no routing of any other device's data? Will it be a heavily loaded router? Will the retry rate in the network make it work harder? What is the cumulative effect of these uncertainties? Does it make battery life drop in half? By 5x? By 100x?

24.3 The Power and Performance Estimator

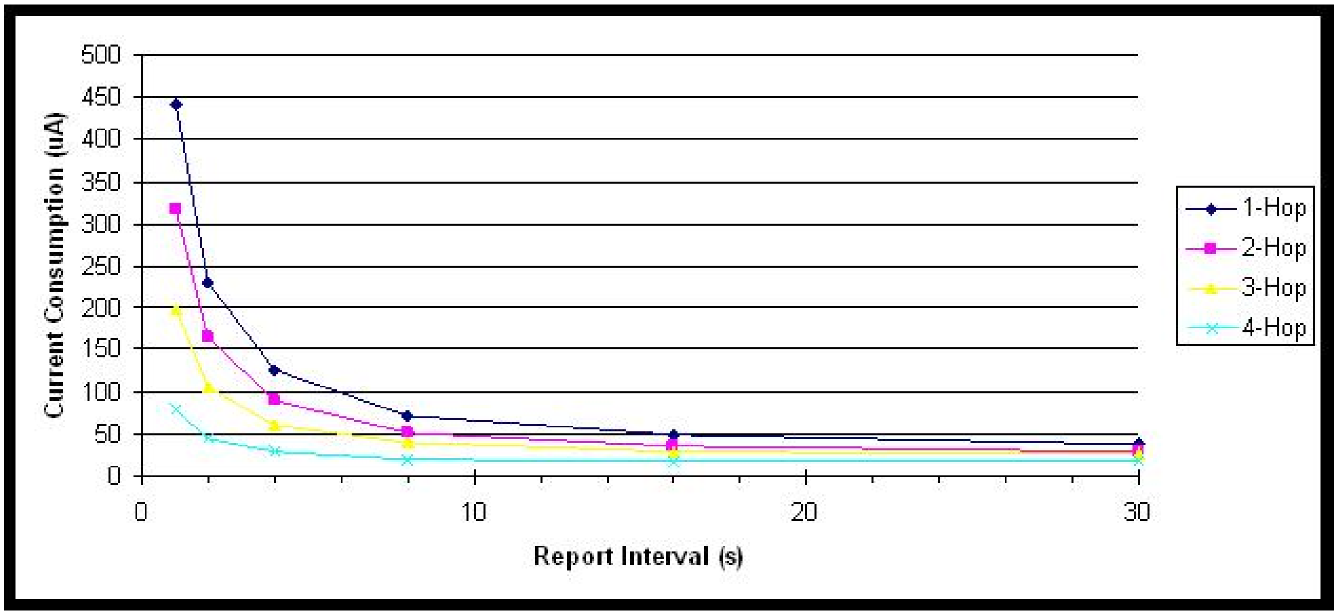
In order to take away some of the uncertainty associated with power consumption and battery life, we've developed the SmartMesh Power and Performance Estimator. You can interact with this Excel spreadsheet to get estimates of power consumption and battery life of motes running in networks that you can specify. The SmartMesh Power and Performance Estimator is an approximate tool for estimating current consumption and battery life, but it is an excellent tool for doing a sensitivity study and for making relative comparisons. The conclusions drawn here are based upon these relative comparisons. In all of the examples, we're using the default values of a 3.6V supply and a room temperature of 25 degrees Celsius.

24.4 Q1: How Does Reporting Rate Affect Power?

Spreadsheet Exercise: Define a network and back off the report rate until you find the minimum possible power consumption for all motes. I'm going to do an IP network with 20 motes, 5 motes at each of 4 hops. I'm going to set the data payload size at 80 bytes, and set the report rate at one packet per second. As a result, the one hop motes are consuming 441 µA. Using a battery like the Tadiran TL4903, which has a nominal capacity of 2400 mA-hr, that translates to a 7 month battery life. The four hop motes, which have no children in the mesh, have about a 3.1 year battery life. Now I repeat at 2 second reporting, 5 second reporting and beyond. The result is as follows:



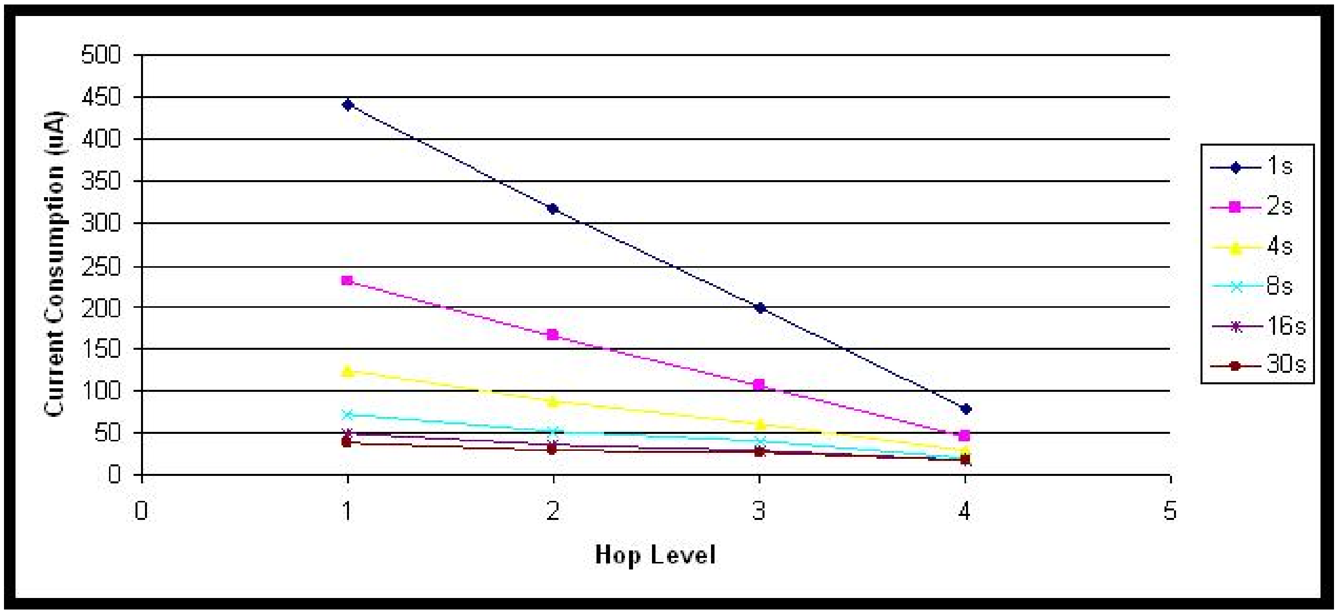
If I back off data reporting from one packet per second to one packet per 2 seconds, I nearly cut current consumption in half, practically doubling battery life. At slow report rates, it doesn't matter much. Whether my sensors report data at once per minute or once per 4 minutes, the current consumption stays flat. If we zoom in on the 1 s through 30 s range we see the following:



It is very easy to build a multi-hop mesh where all devices consume less than 100 µA. At 100 µA, that corresponds to 2.5 year battery life on the TL4903 battery. The conclusion that is important is that if fast reporting is required, ultra-long battery life is difficult. If absolute maximum battery life is required, there is little benefit to reporting data slower than once per minute. Conclusion 1: There is a minimum report rate below which it doesn't matter. In a SmartMesh network, there is some quantity of radio traffic needed to maintain time synchronization across the network. If sensors NEVER send any data, the traffic in the network will be dominated by this time synchronization ("keep alive") traffic. This sets a floor on the minimum possible power consumption in a network. Any time a sensor sends data, that represents a time where this time synchronization traffic need not be sent.

24.5 Q2: How Much Does Routing Cost?

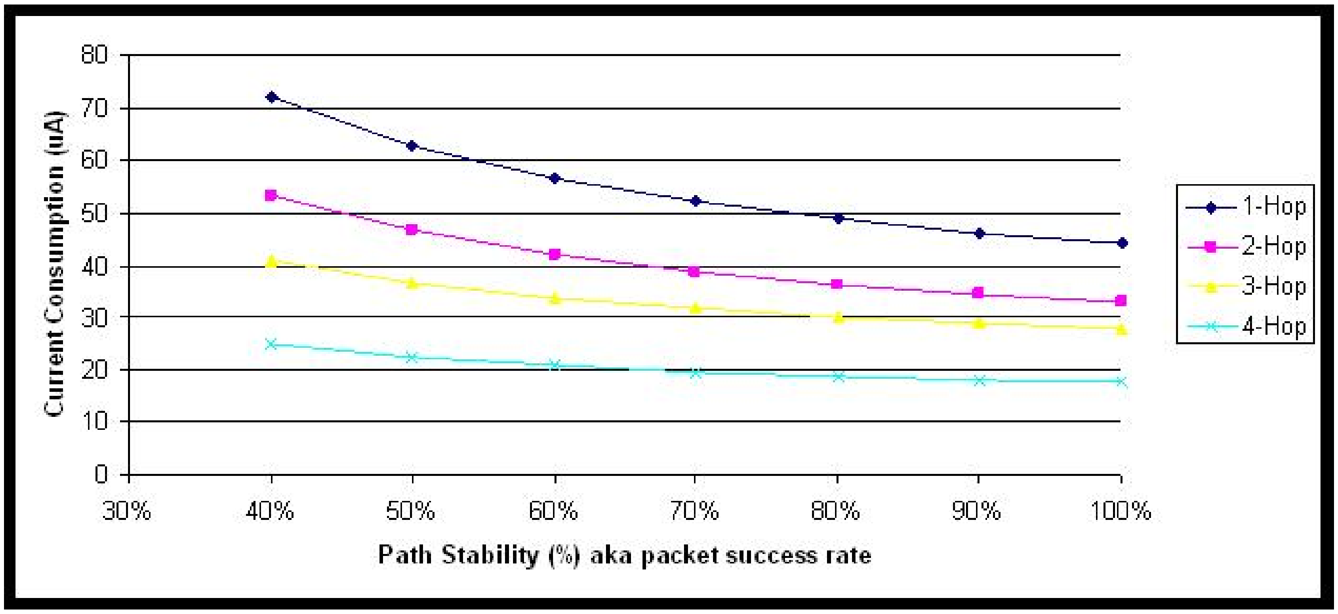
Spreadsheet Exercise: Using the same data set as above, we can show the cost of routing. In this four hop mesh, the four-hop motes only send their own data. The three hop motes send their own plus the data from their children in the mesh. The one-hop motes are responsible for forwarding all the traffic in the network, plus their own. Slicing the data at various report rates we can illustrate the cost of routing.



At lower report rates, the cost of routing data gets flatter. The 4-hop motes are always the lowest power, but the devices doing all the routing are not always at several times the current consumption. At moderate report rates, it is possible to build a network with very flat current consumption and reasonably uniform battery life.
Conclusion 2: It costs something to be a router, but not as much as you might expect, particularly as reporting rates approach the keep alive interval.

24.6 Q3: How Much Does Retransmission Cost?

Spreadsheet Exercise: One of the inputs in the spreadsheet is Path Stability. This means the packet success rate or "one minus the retry rate". Some think that optimizing the radio protocol to reduce bit errors and packet errors is the key to reducing power consumption. Retry rates of ~30% are not uncommon in these networks, but does that mean that batteries are dying 30% faster than they should? The way we illustrate that is by keeping our 20-device 4-hop network reporting at 16 s. I vary path stability. The results are as follows:

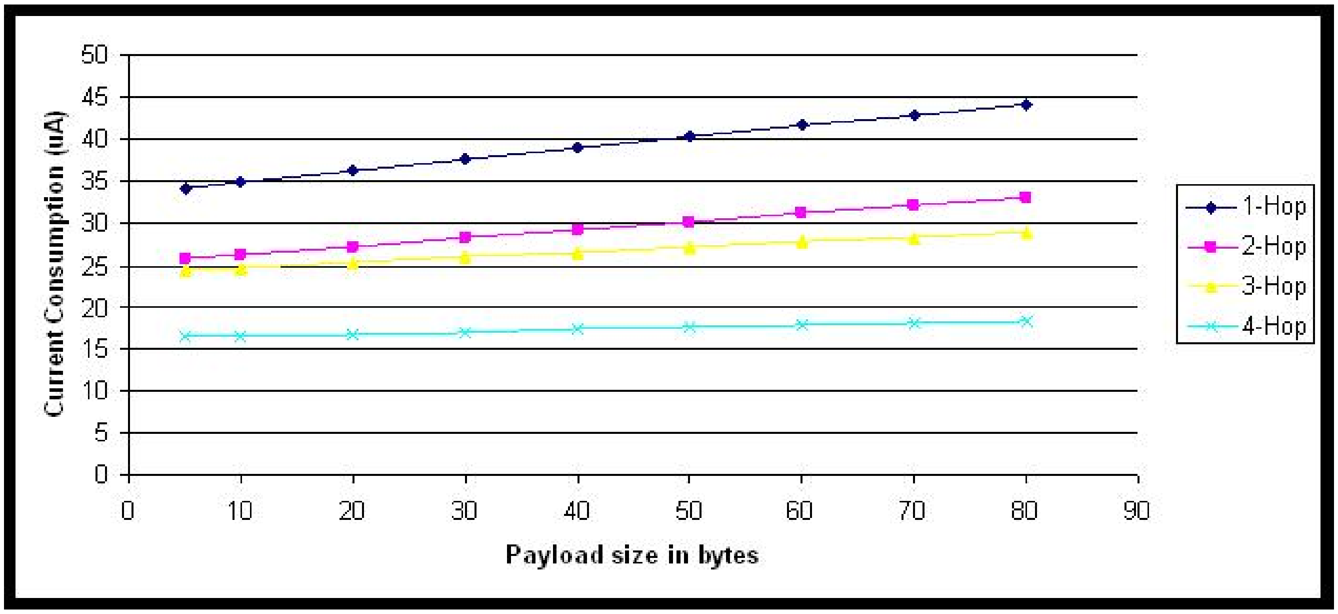


At this moderate report rate, it is a fact that high packet success rate reduces power consumption. At 40% packet success rate, you might think that you will have 2.5x the current consumption and less than half the battery life, but that is not the case. It is certainly better to have strong RF paths with fewer retries needed, but doing a retry should not be considered fatal to battery life. SmartMesh managers understand the tradeoffs between path stability and hop number and optimize accordingly. Sometimes it is better to choose a parent that is closer to the AP than one with higher path stability. This may cause more overall retries in the network, but each packet then has to travel fewer hops to its destination.
We do not expect networks to function well below 40% packet success rate. At very low packet success rates, we find that paths are at the edge of failing completely. The problem there is losing devices entirely from the mesh.

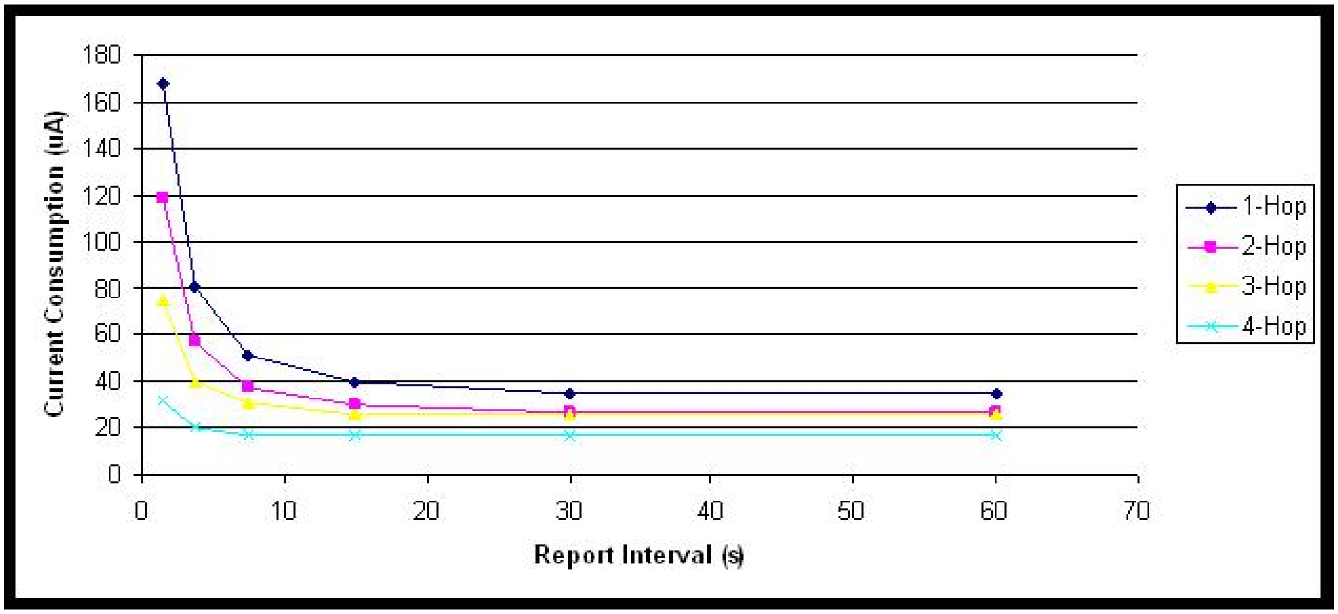
Conclusion 3: Retry rates matter, but not as much as you might think .

24.7 Q4: Can Packet Aggregation Save Power?

Spreadsheet Exercise: In the same 20 mote, 4 hop deep network, we choose one report interval, 20 seconds, and vary the payload size from 5 bytes to 80 bytes. The results are as follows:



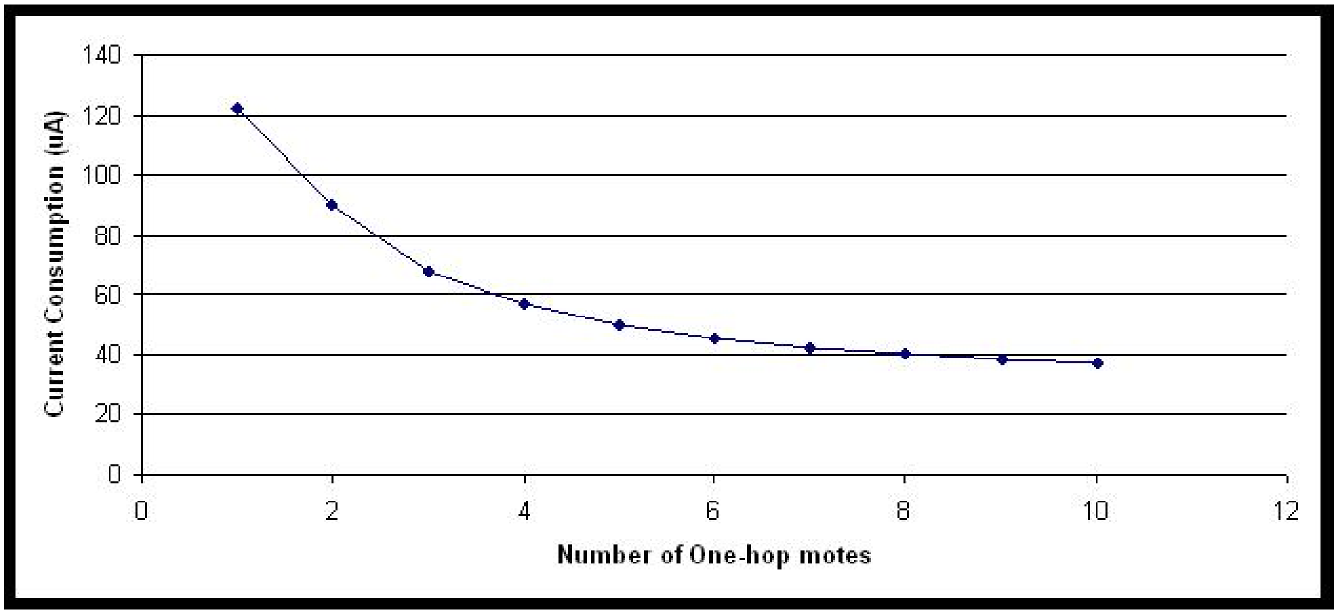
It is true that sending a larger data payload increases power consumption, but sending 18x the data in a 90-byte payload only costs about 10% more in current consumption. Looking at it another way, we consider sensors that all send 80 bytes per minute. One sensor application sends an 80 byte payload once per minute. Another sensor application sends two 40 byte payloads, one each 30 s. The third sensor application sends 20 byte payloads every 15 s, and the fourth sends 10 byte payloads every 7.5 s. The fifth sensor application sends 5 byte payloads every 3.75 s. The sixth sensor application sends a 2 byte payload every 1.5 s. The results are as follows:



All the above are sending the exact same sensor throughput. Sending larger payloads less often is simply a far more efficient way to transfer data. The trade-off of course is latency. Some applications have no use for old data when a newer reading is available. Other applications can use historical data to make very good decisions. There is a ~4x power consumption and battery life benefit that can be obtained if an application can efficiently use the payload capacity available.
Conclusion 4: It is more efficient to send fewer large payloads than it is to send more small payloads. Also, sending a big payload does not cost much more than sending a small payload, since there is a fixed overhead for all packets.

24.8 Q5: How Does Network Depth Impact Power?

Spreadsheet Exercise: We revisit this same 4-hop mesh, but now we vary the number of one-hop motes. Think of it as a 12-mote cluster of sensors at hops 2 through 4, and the 1-hop motes are just repeaters that the customer is reluctantly buying to bridge the gap from the sensors to the gateway. I have the sensors all sending an 80-byte payload once every 15 seconds. We start at 1 one-hop mote, and keep adding them until we get to 10 one-hop motes. We plot the worst case power consumption as a function of one-hop motes. The results are as follows:

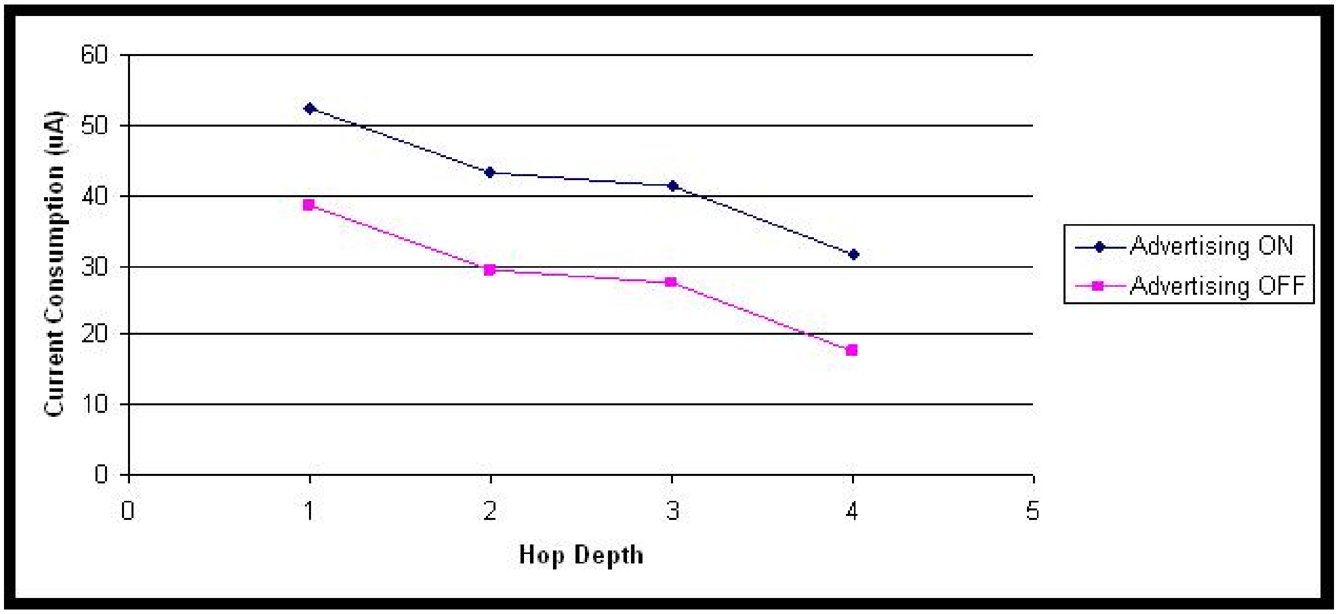


For many systems, the concept of 'battery life' means the moment that the *first* sensor disappears due to a battery dying. At that time, all batteries are replaced. Since the one-hop ring has to forward all the traffic, the first battery to die will almost certainly be in the one hop ring. The easiest way to extend that period of time is to have many one-hop devices. In addition to this effect, having many one-hop devices offers additional robustness vs lost devices. If a network has only two one-hop devices and one is removed, there can no longer be a good mesh at that final hop. Reliability may suffer.
Generally speaking, traffic is increasing at every hop level towards the manager. At each hop level, the motes at that hop level are forwarding all the traffic from descendants, plus adding their own. At each hop level closer to the gateway, the best that can possibly happen in terms of current consumption is for all the devices at that hop level to share the work equally. The more devices there are at that hop level, the lower this equal share can be, and the smaller the penalty if one of these devices goes away.

Conclusion 5: Use as many one-hop motes as possible. Current in any hop ring is inversely proportional to the number of motes in that hop ring.

24.9 Q6: Turn Off Advertising in an IP Network to Save Power?

Spreadsheet Exercise: Note there is an adder for turning ON advertising. In the IP networks, advertising is ON by default. The application must turn it OFF if power savings is desirable. But BE CAREFUL. A new mote cannot possibly join while advertising is off. So, if the application wants to turn it off, it should have mechanisms to turn it back on again. For example, if you form a network, turn off advertising, and then power cycle one mote, it will NEVER join again, unless you turn advertising back on. If the application automatically turns on advertising every time a mote gets lost, then there will be no problem. Also, if the application has a user interface to turn on advertising, then the user can add new devices easily as well. If the application developer is careful, then advertising can be off virtually all the time. This represents a savings of 13.8 µA on every single mote. Looking back at our 4-hop, 20-mote network with 30 s updates, we get the following power curves:



By dropping from 31 µA to 17.2 µA, the leaf nodes could have a battery life of 14.5 years on Tadiran AA cell (2160 mA-hr). Conclusion 6: Turning off fast advertising when you don't need it is valuable, but be careful.

25 Application Note: What to Expect with Motes That Move

25.1 Moving Motes

In current SmartMesh products, a mote traveling beyond the range of its parents needs to reset and rejoin the network with new parents. Similarly, a new mote arriving in the wireless range of an existing network needs to join the network before it can send and receive data. The exact behavior in these conditions depends on which product is being used.

Motes that move continuously around the network and move further than one radio range are not recommended.

25.2 SmartMesh WirelessHART

For SmartMesh WirelessHART, once the network has transitioned out of Building phase to Steady-State phase, advertising is reduced at the motes to conserve energy: the advertising timer is increased from 1.28 s to 20 s on the motes. This means that a mote appearing in range of only mote advertisers during Steady-State will take about 16 times longer to synch up than during the Building phase. AP advertising does not change from the 0.16 s interval, so new motes that appear within range of the AP will not have any different joining behavior at either of the two states. The application can also use the API to speed up advertising again if it is expecting a new mote to arrive.

A mote already in the network that moves has to lose both of its parent paths before resetting. The path alarm timer is 4 minutes in the WirelessHART standard, and getting these path alarms is the mote's trigger to delete a parent path. As such, we expect a moved mote to take a little over 4 minutes to reset. Once the mote resets, the manager will try to reach the mote through its previous downstream links. Once this state machine times out, which could be up to 15 minutes, advertising will be automatically sped up at the remaining motes to search for the lost mote. In the meantime, if the mote hears advertisements at its new locations, it can rejoin the network before the manager detects that it is lost. During periods of fast advertising, either during building or searching for a new mote, Eterna motes use an extra 16 µA and DN2510-based motes use an extra 30 µA.

For more details on the expected time to join, consult the Network Formation section of the SmartMesh WirelessHART User's Guide.

25.3 SmartMesh IP

For SmartMesh IP, advertising keeps the same period unless explicitly deactivated by the application through the API. Each SmartMesh IP frame is about 2 seconds long; motes advertise once per frame and the AP advertises four times per frame. If advertising has been deactivated, no motes advertise at all, meaning that new motes cannot join and motes cannot rejoin the network. We recommend deactivating advertising only when energy conservation is critical and the application is savvy enough to control it without manager intervention. Deactivating advertising saves 14 µA for Eterna motes.

A mote already in the network that moves has to lose both of its parent paths before resetting. The path alarm timer is 1 minute in SmartMesh IP, and getting these path alarms is the mote's trigger to delete a parent path. As such, we expect a moved mote to take a little over 1 minute to reset. Once the manager fails to reach the mote with downstream queries, it will be marked as **Lost** but no automatic changes to advertising occur. Meanwhile, a moved mote is free to rejoin whenever it hears new advertisements.

For more details on the expected time to join, consult the Network Formation section of the SmartMesh IP User's Guide.

25.4 Summary of Differences Between SmartMesh WirelessHART and IP

For this simple comparison, we assume a path stability of 80% and a join duty cycle of 5%.

Parameter	**WH**	**IP**
Time to reset	4 minutes	1 minute
Time to speed up advertising	15 minutes	N/A
Mean Steady-State synch time (1-hop)	60 seconds	188 seconds*
Mean Steady-State synch time (multi-hop, 3 neighbors)	42 minutes	250 seconds*

*If advertising is explicitly turned off by the application in SmartMesh IP, no motes are able to synch up and join the network.

26 Application Note: Migrating Motes Between Networks

26.1 Migrating Motes

Many users will gradually add more motes to a network over its lifetime. At a certain network size, the manager may reach its mote or bandwidth limit and it may be desirable or necessary to move some motes to a new network. This document describes how the customer application can use mote features to intelligently manage this process. We discuss this in the context of a SmartMesh IP Embedded Manager network, but the procedure is equally applicable to a SmartMesh IP VManager network or a SmartMesh WirelessHART network.

26.2 Procedure

A network is deployed in an oil drilling field. As new wells are drilled, motes are added. In the figure shown, network A is nearing capacity, and a second manager is placed near the latest drilling sites to form network B. Since initially there will only be a few motes (grey diamonds) in network B, it is desirable to move a few motes (white diamonds) from network A. Network A and B have a different Network ID.

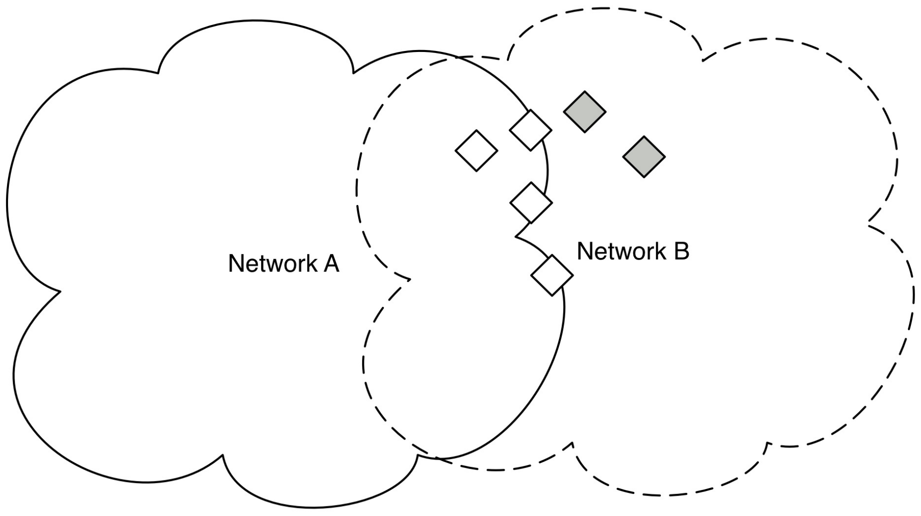


Figure 1 - Moving motes from Network A to Network B

To do this, we make use of the *search* API - this will put the mote into a mode where it listens for advertisements from any network in the vicinity and reports the networks whose advertisements it has heard.

1. The mote is preconfigured with a Network ID - we will call this the "preferred" ID. This is a normal step in commissioning a mote, regardless of whether it is ever expected to move.
2. The sensor application uses the search API to cause the mote to begin reporting information about advertisements heard - Mote ID, Network ID, and signal level (RSSI) are reported. These should be stored by the sensor application.

3. The sensor application sets a timeout for search - this should scale inversely with the *joinDutyCycle* parameter. At 10% duty cycle it should be 2-3 minutes. The longer the search at a given duty cycle, the higher the chance of hearing all networks in the vicinity but also the higher the energy used by the process.
4. At the end of the timeout, if the preferred network (the mote's current Network ID) has been heard at sufficient RSSI (> -85 dBm), the sensor application should issue a *join* command. If the preferred ID is not heard at sufficient RSSI, the sensor application should select another heard network with sufficient RSSI, set the *networkId* parameter to that network, and issue a join command.
5. If the mote tries to join the preferred ID several times but never succeeds, the mote might not have the proper credentials (see Security below) and the sensor application should restart the search process while "blacklisting" this network it cannot join.

As a general practice, this allows motes to join any available network for which they have proper security credentials. The search API can be invoked as a normal part of the joining logic used by the customer sensor application (i.e. whatever is driving the mote API), or in response to an customer application message. In the case of network A & B above, the host-side customer application would change the preferred Network ID on the white diamond motes to B, then reset those motes. Those motes will then use the procedure above to try to join network B, but can safely fall back to network A if they are not actually in range of network B.

26.3 Security

In order to use the Embedded SmartMesh IP Manager with thousands of ACL entries, external SRAM must be installed and the manager must be version 1.4.1 or later.

In addition to hearing an advertisement, a mote must have a join key that matches that used by the manager. Preferred practice is for each mote to have a unique join key, which is placed on the manager's Access Control List (ACL). Only motes on the manager's ACL are allowed to join the network. In the case of network A & B, the white motes being moved from network A would need to be added to network B's ACL prior to resetting them.

27 Application Note: Configuring a Network for Bounded Data Latency

27.1 Increasing Provisioning

With default settings, SmartMesh networks are designed for low-power, high-reliability operation. For typical networks with mean stability over 70% and four or fewer hops, we find that the default settings result in about 90% of packets being delivered within one reporting interval. For example, in a network where motes are each delivering one packet every 10 seconds from their sensors, we expect 90% of these packets to have latency of 10 seconds or less. By adjusting a setting called the *provisioning factor*, we can increase the power used in the network to deliver a higher percentage of packets faster than a given latency target.

In this document, we will be doing the opposite of what is described in the application note "Changing Provisioning Factor to Increase Manager Throughput."

27.2 Restrictions

The settings described in this App Note apply for networks where sensors report at roughly the same rate. For example, in a network where one mote's sensor reports at a 1 second interval and all others report at a 30 second interval, these settings will not necessarily work. For this type of scenario, the Upstream Backbone should be used. For IP networks, see the application note "Using the Powered Backbone to Improve Latency ." For WirelessHART networks, this feature is available in Manager >= 4.1.x.
Networks with low mean stability (below 70%) will have larger latency. In these cases, the provisioning should be increased even higher than the recommended settings in this document. Finally, the settings in this document apply to networks of four or fewer hops. Deeper networks have latency that increases sub-linearly, so a network of eight hops, for example, would meet the same latency targets by doubling the provisioning levels described in this document.

27.3 Provisioning

The default provisioning in all SmartMesh networks is 3x. This means that all motes have at least three upstream links for each expected packet transmission. We have found empirically that this level of provisioning provides the right balance of low power (achieved with fewer links) and high reliability (achieved with more links) for the whole spectrum of networks. So if a mote is transmitting one packet per second, including both locally generated and forwarded packets, this mote will be given three transmit links per second.

There is also a minimum number of links that each mote gets to allow it to reliably keep synchronized to its parents. For some motes, this minimum number is actually larger than the number of links that they would need to transmit data packets alone. For example, each mote in a SmartMesh WirelessHART network gets at least 4 transmit links per 10.24 seconds. If a mote has a one packet per 60 second data requirement, provisioning at 3x means the mote needs one transmit link every 20 seconds. The minimum link number of 4/10.24 is much larger than the 1/20 needed.

27.4 Changing Provisioning

As a rule of thumb, 3x provisioning will get 90% of the packets to the manager "on time". This means that motes reporting at a 10 second interval will have 90% of their packets with latency under 10 seconds and motes reporting at a 7 second interval will have 90% of their packets with latency under 7 seconds. Note that not all customers desire on-time packet delivery, often the objective is simply to deliver all of the generated packets within a reasonable, but less demanding, target.

To boost the on-time delivery to 99%, change the provisioning to 6x.

- SmartMesh WirelessHART: add a line in `dcc.ini` with `TOP_LINK_OVRSUBSCR = 6.0`
- SmartMesh IP Embedded Manager: from manager CLI, type `set config bwmult 600`
- SmartMesh IP VManager: from manager CLI, type `su becareful` followed by `config seti BWMULT=600`

Note that the total throughput of the network in packets per second is halved when you make this change. A network that could support 25 pkt/s with the default settings can only support 12.5 pkt/s with 6x provisioning.

To boost the on-time delivery to 99.9%, change the provisioning to 9x.

- SmartMesh WirelessHART: add a line in `dcc.ini` with `TOP_LINK_OVRSUBSCR = 9.0`
- SmartMesh IP Embedded Manager: from manager CLI, type `set config bwmult 900`
- SmartMesh IP VManager: from manager CLI, type `su becareful` followed by `config seti BWMULT=900`

A network that could support 25 pkt/s with the default settings can only support 8.3 pkt/s with 6x provisioning.

27.5 Power Increases

Additional provisioning doesn't result in any more packets transmitted in the network, just more frequent links to transmit the same number of packets. Leaf nodes will not have any additional power requirement when provisioning is changed as additional transmit links do not come with a power cost. Busy router nodes will be adding several receive links to pair with the additional transmit links, so these nodes will be impacted the most. Again as a rule of thumb, expect busy router power to go up 10-15% when increasing to 6x provisioning and 20-30% when increasing to 9x provisioning.
For low-traffic networks (e.g. 60 second reporting intervals), increasing provisioning to 9x may not actually increase the number of links in the network or any mote's power. In these cases, the minimum number of links is already sufficient to ensure 99.9% on-time delivery.

28 Application Note: Network Coexistence

28.1 Overlapping Networks

All SmartMesh networks operate in the same 2.4 GHz instrumentation, scientific, and medical (ISM) unlicensed wireless band. Here unlicensed means that anyone is free to operate in this band provided that they follow local power limits, but they don't need a license to operate, as they would with TV or cellular phone bands. This band was chosen for its availability worldwide and because there is a IEEE 802.15.4 PHY standard radio specification - thus there are many interoperable radios available. But because anyone can use it, it is crowded with other products - 802.15.4 b/g/n Wi-Fi, Bluetooth, Cordless phones, and ZigBee all operate in the same band. SmartMesh networks are designed with sufficient margin to transmit around potential interference by hopping through channels, under the observation that in real deployments an interferer (or multipath fade) present at channel X at time T0 may not be present on channel Y at time T1. By relying on the average performance over all channels, we see better results than relying on any one channel to be good. Since the other protocols remain on a single channel (or small set of channels), or hop quickly across the entire band (bluetooth), other SmartMesh networks in the vicinity have the potential to be the most deleterious interferers, since collisions can persist when unsychronized networks share a common schedule. This document discusses the features implemented to overcome interference from overlapping networks.

28.2 Collisions in a Single Network

SmartMesh managers schedule activity in such a way to avoid any collisions between transmissions in a single network. The exception to this rule is for "shared" links which appear in three places. Shared links are used for joining motes to contact their join parents. At worst, collisions during this time result in a mote resetting before it becomes operational in the network. In SmartMesh IP, the powered backbone uses shared links to reduce latency. These networks still have sufficient dedicated (i.e. not shared) bandwidth to satisfy all data requirements, so collisions only result in a smaller latency reduction in this case. In SmartMesh WirelessHART networks, motes share the link for sending network discovery packets. The manager assigns timers long enough to reduce the chance of these packets colliding, if they do collide the discovery information is collected safely at a later time.

None of the above specified collision scenarios collide with any regular network traffic as they are scheduled on dedicated channel offsets, and in no other case will there be two transmissions on the same channel at the same time in a single network.

28.3 Avoiding Periodic Collisions

Suppose two networks are located in the same radio space and that both networks have:

- The same current Absolute Slot Number (ASN)
- The same channel list (e.g. default channel blacklist)

- The same lengths for all slotframes
- A transaction in the same timeslot with the same offset

In this case, both networks will have a transmission at the same time at the same channel frequency and, potentially, both messages will interfere with each other resulting in neither succeeding. The packets will need to be retried at a later time. The same scenario will repeat exactly one slotframe later, and it will continue to repeat. The key is that both networks are operating with the same periodicity so a collision that occurs once is likely to occur persistently.

The networks do not necessarily even have to have matching ASN to interfere with each other if they follow the same channel hopping pattern. Networks that are not explicitly synchronized end up drifting and we observe periods where some collisions overlap. These periods, representing the time that the relative drift takes to pass through the length of a timeslot, can last from minutes to hours. If all transmissions for a mote match the pattern of transmissions in an overlapping network, this mote can end up unable to communicate for long enough to lose contact with its parents and reset. If the occasional transmission succeeds, the mote will stay in the network - it is only when all data and keep alive packets are persistently clobbered that the mote will be lost.

The amount of overlapping between two networks is very low. Each network collects data through a single Access Point (AP) node, and each AP can only listen or transmit on a single channel in each timeslot. Compared to the 15 channels that are available to SmartMesh networks, we expect a maximum of 1/15th of the overall timeslots to contain actual transmissions at the busiest location in the network. Furthermore, the transmissions don't fill the entire length of the timeslot so the fraction of the total bandwidth occupied is even lower. This suggests that there is plenty of room for motes to safely coexist in the same radio space providing we can avoid periodic and persistent collisions. If collisions occur, but happen randomly instead of consistently, this manifests as an overall decrease in path stability in both networks which doesn't cause any serious problems.

SmartMesh networks avoid periodic collisions differently in the IP and WirelessHART product families.

28.3.1 SmartMesh WirelessHART

Each SmartMesh WirelessHART network has the same slotframes with fixed lengths. The upstream slotframe is 1024 slots, the downstream slotframe is 256 slots, and the advertisement slotframe is 128 slots. Changing these slotframe lengths is not allowed, so we need a different mechanism to avoid periodicity.

Upstream communication is randomized by giving each mote a minimum of four transmit links. The offsets for these links are chosen completely randomly and there is a little jitter in the timeslot chosen for each upstream link. The chance that any four links from network A collide with network B is thus less than 1 in 15^3 ,i.e. 1: 3375.

Downstream communication is randomized by giving the AP both broadcast and multicast links and by using two source routes for each mote target. If the first attempt to reach a mote downstream fails, it will be retried with the first hop transmission on a different timeslot and relatively random timeslot, and the same holds for each subsequent transmission along the multihop route.

Advertisement and network discovery transmissions happen on a timer instead of in every assigned slot. This means that these transmissions will never occur with perfect periodicity and shouldn't cause persistent problems in neighboring networks.

28.3.2 SmartMesh IP Embedded Manager

SmartMesh IP Embedded Managers are memory-constrained so we do not have the ability to assign several links to each mote. To overcome this limitation, we assign fewer links in shorter slotframes which does increase the chance of persistent collisions. However, since the IP networks have a single base slotframe length, this is easy to randomize. At network boot-up, the manager chooses a random slotframe length between 256 and 284 timeslots which it uses for all upstream, downstream, and advertising activity.

The default minimum number of upstream links for an IP network is two. If there aren't many motes in the network and power isn't a huge concern, this parameter can be increased in the "ini" settings to provide more immunity against persistent collisions.

Downstream communication is randomized by giving the AP both broadcast and multicast links and by using two source routes for each mote target. If the first attempt to reach a mote downstream fails, it will be retried with the first hop transmission on a different timeslot and relatively random timeslot, and the same holds for each subsequent transmission along the multihop route

28.3.3 SmartMesh IP VManager

SmartMesh IP VManagers use fixed slotframe sizes for smaller networks. In the default configuration, both the upstream and downstream slotframes are 512 slots long and multiple links are assigned in both directions on each path to decrease the chances of persistent collisions. As network sizes increase, the VManager needs to reassign cells to multiple devices, so motes monitor for bad links upstream and report this information to the manager. As networks grow into the thousands of motes, VManager introduces randomization into the Advertising and Discovery frames.

28.3.4 Mixed IP - WirelessHART Environments

The IP and WirelessHART solutions operate with different timeslot lengths, 7.25 ms and 10 ms, respectively. This alone makes them safe to operate in the same radio space without fear of persistent collisions between them.

28.4 Clear Channel Assessment

All SmartMesh products feature optional Clear Channel Assessment (CCA). When enabled, all devices in the network will listen for a short period before transmitting, and abort the scheduled transmission if another device's transmission is overheard. This is intended for coexistence between neighboring unsynchronized networks - it has no effect in-network since we avoid collisions locally as described above. Use of CCA is off by default - the threshold for aborting transmit as defined in IEEE 802.15.4 is low, and networks that could operate successfully may be completely blocked by a moderate strength wide band interferer. We recommend that you only consider using CCA at the request of another nearby 802.15.4 network operator, and then only if they are experiencing coexistence problems.

28.5 Empirical Results

In our test lab, we routinely have about 40 SmartMesh networks with about 900 motes operating continuously. During times of peak traffic, we have captured upwards of 20 packet/s on each available channel in our building. This environment should be more challenging than anything faced by customer deployments and we rarely see motes reset due to collisions.

We ran a dedicated test with 1000 motes operating in our small lab split into 8x125-mote WirelessHART networks. At its peak, the total traffic was 181 packet/s in this network and the networks maintained better than 99.9% reliability. Compared to a low-traffic setting in which the motes kept synch but didn't send data, the path stability dropped from 80% to 68%. Overall, this results in increase latency and power for the network, but didn't result in any serious problems.

29 Application Note: How to Choose a Join Duty Cycle

This document walks through the decision for how to set the *join duty cycle* on the mote. This is normally done by the sensor application that talks to the mote.

29.1 Background - What is the Join Duty Cycle?

When you send the mote *join* API command, the mote starts searching for a network to join. Searching means it listens on a single channel for a while, and then sleeps for a while, and then resumes listening on a different channel. The total period (listen time + sleep time) is 3-4 seconds long (depending upon SmartMesh family). The join duty cycle is a one byte field that can be set to anything between 0 and 255, to set what portion of this period is spent listening. The greater portion of the time your mote listens the faster that mote will hear an advertisement and get joined to the network, but at increased average current consumption. For example, if you set the search duty cycle to 255 on a mote, it will listen constantly, changing channels every few seconds. If you set the join duty cycle to 26, it will spend about 10% of the time (26/255) listening and 90% of the time sleeping. This will consume much less power, but will hear an advertisement much more slowly. Thus join duty cycle setting gives you a tradeoff between average power and time spent searching. The expected time for a mote to synchronize to the network is a function of join duty cycle and network topology via the number and rate of advertising neighbors:

sync time = advertising rate per device * # channels / (# of devices advertising * path stability * listener join duty cycle)

In the SmartMesh starter kits (DC9000 & DC9021 and DC9007), the motes operate in **master** mode. They join on their own, with no commands from an external processor. This mode is typically used for demonstrations. In contrast, most real applications operate the mote in **slave** mode, where the mote needs to be told to join. Along with the join command the mote should be told what duty cycle it should use for searching - this document articulates the tradeoffs between "fast enough" network formation time and average power in a number of network scenarios. In SmartMesh WirelessHART the mote returns to a default when reset or power cycled, so if something other than the default value is desired, it must be set at each boot. In SmartMesh IP, the join duty cycle persists through reset and power cycle and only needs to be set once.

29.2 What Join Duty Cycle Should I Use?

There are four common approaches to setting the join duty cycle:

1. For powered devices: just set it to 100%. The benefit to this is that this is always the fastest to join. If you place a mote in a position where it cannot hear any neighbors the device will burn ~5 mA searching and never hearing any advertisements.

2. For scavenger devices: set it to the power level you can afford. A scavenger device can only guarantee a limited amount of current. Set the search duty cycle to fit in that budget. For example, if you have a scavenger that can source 200 µA continuously to the mote, and the mote consumes 5 mA in receive, then we know we have to set the search duty cycle to 10 (10/255 = 4%) or less. The mote will join more slowly than it would at a higher duty cycle, but it will stay under the power budget for as long as it needs to search.
3. Set it to match your battery life targets. If the device has a strict battery life target, then it has a specified average current budget, and you can calculate a join duty cycle as in #2 above. E.g. If you have a 2000 mAhr battery, and the target lifetime is 10,000 hrs, then you can afford 200 µA. That way the device will meet the battery life target whether it is in the network or not. This is also the approach for devices that will commonly be left in a "stranded" position but is expected to join whenever a network becomes present.
4. Try to pick a value that is "good enough" in speed and in power. If you were to test for yourself, you would find that the total time it takes a dense network (one where most motes can hear 8 or more neighbors) to fully form is only weakly associated with joined duty cycle. If it takes 30 minutes for a 100 mote network to form at 100% search duty cycle, it may only take 32 minutes at 50% and 35 minutes at 25%. By setting the search duty cycle to a lower value, you are only a little bit slower, and you save a lot of power on the "stranded" device. If you go too low, well under 10%, then things can slow down a great deal. In the limit, a search duty cycle of 0 (0.2%) will join very very slowly and will consume < 10 µA average current. If your device is not powered and is not a scavenger, and is not likely to be left stranded, so 25% is an excellent place to start. Confirm things are fast enough for your installers and consider adjusting this value based on that feedback.

29.3 The Sensor Application State Machine

SmartMesh IP motes persist the join duty cycle setting, so normally this would be set in manufacturing or when initially commissioning the device.

For SmartMesh WirelessHART, since join duty cycle is not persisted, you should consider the following rules for your application:

Rule #1: Your software should always be ready to get the Boot Event.

This event is sent by the mote whenever the mote resets. If it is power cycled, it sends the boot event. If the mote becomes lost from the network, it sends the boot event. If the mote receives a reset command over the air, it sends a boot event. The simplest state machine is to set the join duty cycle each time a boot event is received.

Rule #2: Your software should always be prepared to set the join duty cycle before it sends the join command.

Even if it turns out that you like the default join duty cycle, that default may change someday. You should always write the value you want to use (best practice is to read the current value and write a new one if different from the desired value).

29.4 Summary

To repeat, the process will be:

- Step 1: Set the join duty cycle using the mote API command in your start up routine. Make sure you call this command anytime you are going to send the join command.
- Step 2: As a placeholder, use the value 64 (25%).
- Step 3: If you don't mind consuming ~5 mA constantly, set it to 255 (100%) instead.
- Step 4: If your device is on a strict power budget, use that to calculate what join duty cycle you can afford.
- Step 5: Test your value empirically and decide if it needs to be tuned.

30 Application Note: SmartMesh Security

30.1 Introduction

Wireless packets are transmitted through the air where anyone can theoretically eavesdrop. Indeed, there are so-called *packet sniffers* that can listen to all 16 channels in the 802.15.4 spectrum (the 2.4 GHz ISM band) at the same time, so channel hopping alone is not sufficient to protect data from outside listeners. Security protocols must be designed so that a listener hearing the raw bits of every single packet still cannot decrypt any of the information. All of our products have the same security features as part of the standard product. We believe that all customers need secure networks whether or not they recognize it, and we have built SmartMesh products to always use secure communications. The only downside to security is that a small amount of overhead is added to each packet transmission, but the incremental power consumption this incurs is trivial. The security standards implemented in Dust products are industry best practices, and while Dust's implementation is thorough, it is not novel (that's good!).

30.2 Goals

A secure wireless network must have the following properties:

- **Message Integrity**—Data received at the destination should not be accepted if it has been modified in transit. This is an end-to-end property that must be maintained even in the presence of a malicious router and even when the packet goes through many hops from source to destination. This is also called *Authentication*, as it is intended to confirm the identity of the sender to prevent *Man-in-the-Middle* attacks where each side in a conversation is unknowingly talking to a third party.
- **Access Control**—Motes should only accept data from authorized motes. This is an end-to-end property, though it also has a link-layer corollary. Data from unauthorized motes should not be permitted to result in a denial of service (DoS) attack.
- **Confidentiality**—An eavesdropper that intercepts any encrypted data should not be able to determine anything about the plaintext data except the plaintext length (semantic security).
- **Replay Protection**—If an adversary captures legitimate encrypted traffic and re-injects it into the network (possibly at a different location), that traffic must not be accepted at the destination without detection. This is an end-to-end and a link layer property.
- **DoS resistance**—It should be difficult to inject packets into the network, congesting it to a point that prevents the network from operating normally.

30.3 SmartMesh Security Features

- **Message Integrity**—Message integrity is achieved with two 32-bit Message Integrity Codes (MIC), one at the link layer, *i.e.* at each hop, and one at the network/mesh layer, *i.e.* end-to-end within the mesh. The link layer MIC enables the receiving mote at each hop to confirm that the packet is being transmitted by a manager-approved mote. The end-to-end MIC is used at the destination to both decrypt and authenticate the packet. This MIC guarantees that the packet was not altered at any hop after the sender transmitted it. Both MICs are calculated via the CCM* algorithm (see below).
- **Access Control**—The manager maintains an Access Control List with a list of devices allowed to join the network, along with their join keys. A device cannot join without presenting the correct 128-bit Join key.
- **Confidentiality**— Confidentiality is guaranteed with a CCM* stream cipher based on AES 128-bit encryption of the payload. Encryption keys are shared secret Session Keys that are randomly generated and securely delivered at runtime. No eavesdropper outside the network can decrypt any payload. No mote inside the network can decrypt any other mote's payload.
- **Replay Protection**—In joining the network, each mote encrypts the first message using a persistent join counter – any replay join requests are detected and dropped by the manager. After the mote joins the network and obtains a session key, all packets are encrypted using a monotonically increasing 32-bit nonce counter. At the destination, the receiver tracks a history of received nonce counters to eliminate old and duplicate packets. In addition, link-layer MICs described above are computed using a timestamp based nonce. Replays are detected and dropped.
- **DoS resistance**—The security attributes that achieve Replay Protection also provide DoS resistance.

30.4 Encryption and Authentication

All data link layer (DLL) messages are authenticated with CCM* with AES-128 in hardware. This is done with join requests using a well known key, and with all other messages using the run-time Network Key. The DLL nonce counter is based on shared time (ASN) to prevent replays. Network Layer headers are authenticated and payloads authenticated/encrypted using CCM* with AES-128. The join messages use the appropriate symmetric Join Key as described above. Motes persist their nonce counter through reset. Run-time Session Keys as described above are used for all other end to end session traffic. Each session carries its own nonce, which mitigates against replay and man-in-the-middle attacks.

Message integrity and replay protection are ensured through time-based message authentication at the link layer. Session based authentication and encryption at the network layer ensure confidentiality, source authentication, and replay protection.

30.4.1 Keying Model

In SmartMesh networks, each mote contains a 128-bit Join Key stored securely in non-volatile (NV) flash. The mote authenticates itself to the network by sending a join request encrypted with this Join Key and Join Counter. The network manager decrypts this message to confirm it was sent by a mote possessing the correct key and with the expected join counter. If authentication fails, the join request is dropped.

After device authentication, the manager distributes the following Session Keys

- **Manager Unicast Session**—This session is used to securely transport all keys, and most network configuration messages. The mote communicates with other motes on a specific schedule. All commands associated with the creation and maintenance of that schedule are strictly restricted to the manager unicast session (*i.e.* only the manager can lay out network resources).
- **Manager Broadcast Session**—This session uses a single key for all motes and is used for commands that are destined for all motes simultaneously. This session is used for over-the-air-programming (OTAP) and controlling advertising rates in the network. Note that an OTAP image can only be activated using the manager Unicast Session.
- **Gateway Unicast Session**—This session is used to encrypt and decrypt sensor payloads. The manager and mote perform that encryption and decryption function.
- **Gateway Broadcast Session**—If the gateway ever needs to broadcast a payload to all sensor processors, this key will be used to encrypt and decrypt that payload.

In SmartMesh IP, only one session is used for both management and data traffic.

Each of the above security sessions has a key. So in an N-mote network, there are N manager unicast session keys, and N gateway session keys, plus the two broadcast session keys. The final key is the *network MIC key*. This shared key is used by all motes to authenticate packets at the Data Link Layer. In other words, a routing mote uses this key to confirm the packet is from a valid neighbor, without possessing the key needed to decrypt that packet.

30.4.2 Manager Security Policies for Joining

The manager has three security policies for joining that can be chosen by the network administrator.

- **Accept Common Key**—In this security policy, the manager will grant network access to *any* mote that presents a network wide shared *join key*.
- **ACL with Common Key**—In this security policy, the manager will grant network access only to motes whose globally unique 8-byte MAC address are on the access control list (ACL) *and* present the network wide shared join key. Join requests that present an invalid MAC address or the wrong joinkey will be dropped.
- **ACL with Unique Keys**—In this mode, each mote on the ACL has a unique join key. A device will be granted access to the network only if it presents the correct MAC address and join key. All other requests will be dropped. **This is the recommended and most secure mode of operation.**

30.4.3 Key Management

Key generation and distribution is executed by the network manager. Key generation is based upon NIST specification SP800-90. CTR-DRBG samples thermal noise and an XOR function to generate a large random seed. There is no automatic key rotation policy in the network manager, but the network manager does serve up key rotation APIs to invoke the secure generation and rotation of any keys.

30.4.4 Passwords

Since password defaults are published in documentation, it is strongly suggested that users change their passwords.

SmartMesh WirelessHART Managers and SmartMesh IP VManagers have a number of passwords that are user configurable:

- Linux login username and password
- Linux root password
- CLI (console) username, password & access level
- Admin Toolset password (WirelessHART only)
- API username, password & access level

SmartMesh IP Embedded Managers have two login levels: user (full access) and viewer (read only access). Both user passwords can be changed.

30.4.5 A Note on CCM*

CCM* stands for **C**ounter mode with a **C**BC-**M**AC. Wasn't that helpful? CBC means a **C**hain **B**lock **C**ipher – AES is a block cipher – it works on a 16-byte block of data. To make is useful for arbitrary length longer pieces of data (a *stream*), we use a technique called *Block Chaining* – using the output of one AES block computation as an input to the next block, so that the security of all the blocks is "chained" together. In CCM* we also generate a **M**essage **A**uthentication **C**ode, also called a Message Integrity Code or MIC that's used to verify that the data wasn't changed. The * in CCM* means that you can do an authentication operation, an encryption operation, or both. CCM uses an incrementing counter as a *nonce* – a "number used once" as an input to encryption/decryption operations. While it is not necessary to keep the nonce a secret, both sides need to know what the current nonce is (along with the key) to properly decode the packet, and the number must only be used once. For the link-layer nonce, the motes use the ASN – the count of slots elapsed in the network. At the network layer this is an incrementing message counter.

31 Application Note: Using the TestRadio Commands

31.1 The testRadio Commands

The SmartMesh WirelessHART Mote firmware contains API commands to measure the quality of a wireless link between two motes in a standalone test: *testRadioTxExt* and *testRadioRx*. These commands exercise low-level features of the radio and can only be used on motes which have not joined a network. They allow an integrator to expose radio testing commands programmatically through their application front end, as opposed to exposing the device CLI.

These commands are typically used to verify that an antenna design/manufacturing behaves as expected. This application note details how to use these commands to perform a Packet Delivery Ratio (PDR) test between two motes. This would typically be done for top-level assembly test or field debugging.

You will use the *testRadioTx* and *testRadioRx* commands to send 1000 packets from a transmitter mote to a receiver mote, and gather statistics on how many packets were received.

31.2 Setup

You need the following:

- 2 SmartMesh WirelessHART Motes or SmartMesh IP Motes which are not connected to any network
- A computer with 2 available USB ports and the SmartMesh SDK installed
- 2 USB cables

Before starting, write down the serial port numbers corresponding to the API port of the two motes connected to your computer.

31.3 Running the Experiment

- Start 2 instances of the APIExplorer application (part of the SmartMesh SDK).
- Connect both applications to the API port of your two motes.

- On the APIExplorer connected to the transmitter mote:
  - Select the *testRadioTxExt* command and populate the fields as follows:
    - **testType**: packet - This will instruct the mote to transmit IEEE802.15.4 compliant packets.
    - **chanMask**: 8000 -This indicates that you want the packet to be sent on channel 15 (`0x8000` corresponds to the bitmap `b1000000000000000`, i.e. only bit 15 is set), or 2.480 GHz.
    - **repeatCnt**: 1 - Number of times to repeat the packet sequence.
    - **txPower**: 8 - The mote will send the packets at +8 dBm conducted power.
    - **seqSize**: 1000 - Number of packets in each sequence
    - **sequenceDef**: pkLen = 125, delay = 20000. Packets are spaced by 20 ms, so sending 1000 packets will take 20 s.The mote will add two bytes of CRC at the end, resulting in a maximum-length IEEE802.15.4 packet.
  - Do not click on the **send** button yet.
- On the APIExplorer connected to the receiver mote:
  - Select the *testRadioRx* command and populate the fields as follows:
    - **channel**: 2.480 GHz. This corresponds to channel 15, the channel the transmitter is transmitting on.
    - **time**: 30. This instructs the mote to listen for packets for 30 s. The transmitter will transmit for 20 s so this provides a little buffer.
- Start the test:
  - Click on the **send** button on the receiver side. The mote starts receiving for 30 s.
  - Click on the **send** button on the transmitter side. The mote starts transmitting 1000 packets, for a total duration of 20 s.
- Wait 30 seconds and collect the results:
  - On the APIExplorer connected to the receiver mote, enter the *getParameter* command, and the *testRadioRxStats* subcommand. Press **send.**
  - The response fields contains the following fields:
    - **rxOk** is the number of packets received successfully.
    - **rxFail** is the number of packets received, but for which the CRC fails. This indicates the packet was received, but got corrupted during transmission.

31.4 Interpreting the Results

A number of physical phenomena can cause packets not to be received correctly: interference, multi-path fading, improper antenna connection, sender and receiver being too far apart, etc. In these cases, packets can be received at a low SNR, leading to either the packet being corrupted (and appearing in the **rxFail** counter), or not being received at all.

You should hence determine:

- How many packets were received correctly? This is the **rxOk** counter.
- How many packets were received, but corrupted? This is the **rxFail** counter.
- How many packet were lost? This is the difference between the number of packets sent and the sum of the **rxOk** and **rxFail** counters.

32 Application Note: Best Practices to Limit Average Current During Peak Periods

The LTC5800 is intended for low-power designs - but while normal mote operating currents of < 50 µA are typical, for short intervals (such as when the part is booted) current may be much higher. This poses challenges for designs that run off of a current-limited supply. The LTC5800 is designed to limit itself to an average current of 360 µA averaged over a 7 s window during these peak-current intervals, however certain system level considerations must be respected to meet this current target. This current level corresponds to the low-current boot mode in legacy SmartMesh WirelessHART products (DN2510), which was intended to support operation of the mote on a 4-20 mA current loop.

If not specifically highlighted, each item applies to all SmartMesh families.

- **Reset** – boot consumes ~800 µA and takes 800 ms. Repeatedly resetting the part after boot could raise the average current above target. It is our guideline that the OEM processor not reboot the part more often than every 2 s.
- **Join Duty Cycle** – The join duty cycle should be set to no more than 5% to ensure the average current meets this target.
- **UART activity (SmartMesh WirelessHART)** – UART traffic has a small impact on current at 9600 bps. Continuous looping on an API at 115 kbps will raise the average current by ~75 µA and should be avoided, but this is not a normal use case. Normal use of the part makes use of notifications instead of polling. The time pin should not be polled more often than every 2 s.
- **Persistent Parameters** – The system allows the user to change Flash-based nonvolatile parameters via the *setNVParameter* API – however, consecutive writes can exceed average current target. Our guidelines are:
  - Wait 2 s after receiving a boot notification to do a NV write
  - Consecutive NV parameter writes must be spaced at 2 s minimum
  - If NV writes are done after the boot notification, wait 2 s before issuing a join request
- **OTAP** – OTAP requires erasing the file system (to store the incoming OTAP file) and main Flash area (to replace the running firmware). In the current release (SmartMesh WirelessHART 1.0.2, SmartMesh IP 1.2.0), this is done atomically and will exceed the current target, so OTAP should be avoided with this version of code on current limited devices. OTAP erase will be duty cycled to meet the current target in a subsequent release. OTAP is an infrequent activity, and many systems are never upgraded in their lifetime.
- **Powered Backbone (SmartMesh IP Embedded Manager, SmartMesh WirelessHART Mgr >=4.1)** – The low-current boot mode of the DN2510 will not meet the power target with this networking mode. LTC5800 motes can operate as non-routing leafs on the backbone in both IP and WirelessHART networks.
- **Severe congestion** – The manager limits mote links such that if every link was used, the current target would not be exceeded. In practice, only a fraction of these links are used, so the current release does not account for infrequent housekeeping operations such as radio retune due to a temperature change. In extremely rare cases, a severely congested mote could exceed the target if the device needs to retune its radio. Should this result in a reset, the mote will retune at boot and join normally.

- **Scratchpad use (SmartMesh WirelessHART mote >= 1.1)** – Starting with version 1.1.x mote software, a scratchpad is available for customer use. The guideline is the same as for persistent parameters - Scratchpad writes should be spaced by at least 2 s to ensure the average power target is met.

33 Application Note: Methodology For Pilot Network Evaluation

33.1 Summary

There is often a desire to deploy a pilot network, test it, and analyze to see if it will perform adequately in an environment representative of those in which it will ultimately be deployed. The closer the pilot deployment matches the real one with respect to mote/sensor locations and topology, their report rates, etc., the better the user gets a feel for how the real network is going to perform.

The SmartMesh SDK provides simple tools evaluating a network in a user specific environment. Here we outline a step-by-step approach that will help the user deploy the motes in a specific environment for testing. After deployment, the utilities provided with the SDK may be used to configure the different motes in the network to report data at the desired rates, thereby emulating the actual sensor reporting behavior in a real deployment. This document describes the methodology to gather statistics from the deployed network over a period of time, and analyze these statistics to evaluate the performance of the pilot network with respect to key parameters such as reliability, data rates, bandwidth, etc.

33.2 SmartMesh Starter Kits and SDK

The SmartMesh Starter Kits comes standard with 1 manager and 5 motes. Additional motes may be purchased in order to grow the network size, if required. The user needs to decide the appropriate product family for their use - SmartMesh IP or SmartMesh WirelessHART, based on prior discussion with a Linear FAE. If choosing SmartMesh IP, this document covers using the starter kit with an Embedded Manager, not the VManager. All the documentation, software utilities and tools are available for download at http://www.linear.com/starterkits.
The SmartMesh IP Easy Start Guide or SmartMesh WirelessHART Easy Start Guide go over the installation of starter kit hardware and basic function of the SDK software. The rest of this application note assumes that the appropriate SDK has been properly installed on the user PC and the user is able to work with CLI (Command Line Interface) on the manager.

⚠ More detailed installation instructions can be found in the SmartMesh IP Tools Guide or SmartMesh WirelessHART Tools Guide.

33.3 Evaluation Methodology

These the basic 5 steps for evaluation a pilot network-

- Step 1 - SDK Installation on a PC

- Step 2 - Pilot Network Deployment of the SDK Motes and Manager
- Step 3 - Configuring Motes for Specific Data Rates
- Step 4 - Gathering Data and Statistics
- Step 5 - Analyzing Data

33.4 Step 1: SDK Installation on a PC

1. Download the SDK from http://www.linear.com/starterkits. Also download appropriate documentation based on the desired product family - SmartMesh IP or SmartMesh WirelessHART.
2. Review the Easy Start Guide document and install the SDK on your PC.
3. Make sure that you can connect the PC to the manager and exercise the CLI.
4. For SmartMesh IP make sure that Serial Multiplexer is installed and running properly. Detailed installation instructions can be found in the SmartMesh IP Tools Guide.

33.5 Step 2: Pilot Network Deployment of the SDK Motes and Manager

- Review the application note "Planning A Deployment". The key deployment guideline is that each mote be placed within range of 3 other neighbors/motes.
- Determine range based on the physical environment. You may start with a range of 50 meters and later adjust it based on your results.

> For the analysis of your network it will be extremely helpful if you are able to document the locations of each mote you place, or at least the distance to their neighbors. This distance data will be crucial for making plots against RSSI once you have run your network for a while.

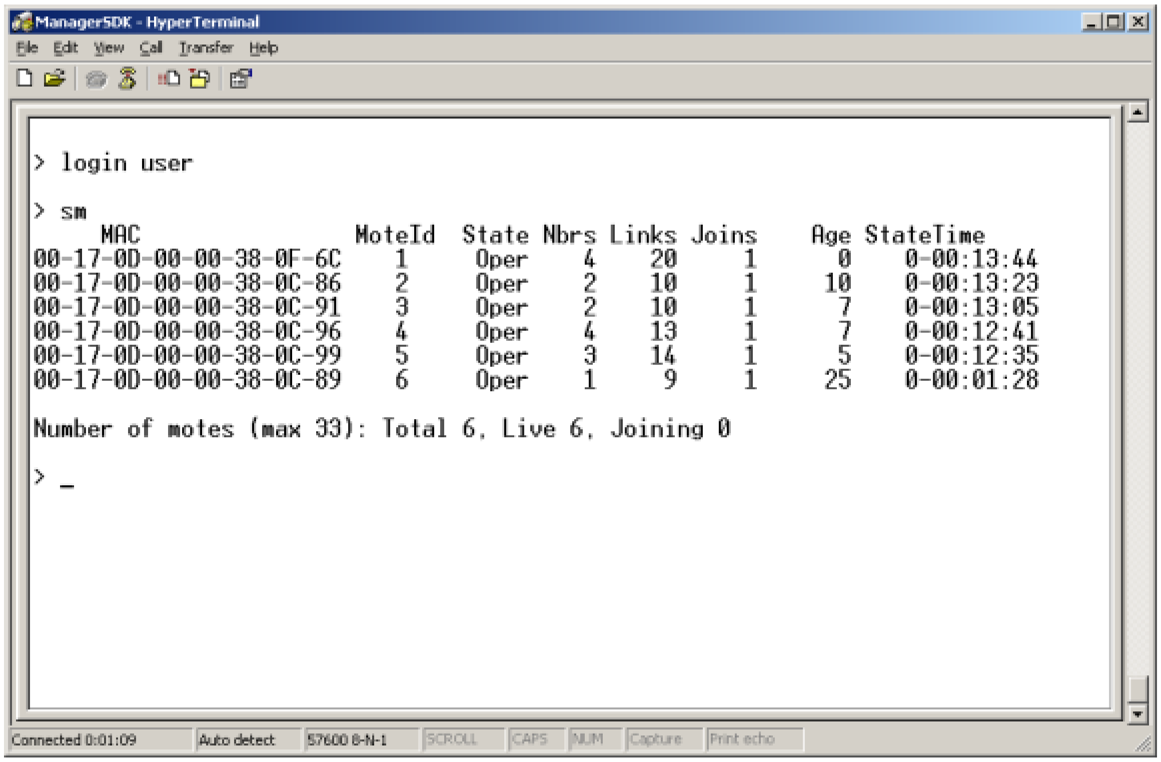
- Based on the above 3-neighbor guideline and the area you want to cover, determine the number of motes you will require for a pilot network deployment. Make sure you have the required number of motes for a successful deployment.
- Deploy the manager first. Make sure you have access to it via your PC. Turn it ON and make sure you can exercise the CLI commands.
- Make sure that the motes have fresh batteries. They ship with fresh batteries but during install and testing of the SDK, they may have been drained. This can happen if the mote is left on without the manager/network being available. In this case the mote keeps searching for a network with the receiver on 25% of the time which might deplete the capacity.
- Start by placing the motes where you are likely to place the sensor for monitoring specific parameter. Turn the power switch ON. For SmartMesh IP motes, if enabled, the Status_0 LED will begin blinking.
- Note the MAC Address of the mote and the location it is deployed appropriately in a map or a table as desired. This will be helpful when you will configure the mote for data rate, in Step 3.

Before placing the motes, make sure they are configured in **master** mode. This is their default setting, but during testing of the SDK you may have changed their configuration.

- Follow the methodology outlined in the planning Application Note to place additional motes, if required, which will act as repeaters.
- For SmartMesh IP, once all motes are deployed, the two Status LEDs will be lit, indicating that each mote is now connected to the manager.
- Now go back to the manager to check that the network has formed. You can do this via the manager CLI connection. Use the command `sm` (short for "show motes") to display a list of the motes in the network.

It can take several minutes for the network to form once deployed, depending on the size of the network. If a mote doesn't seem to join, try restarting it. You can also trace mote state changes with CLI command `trace motest on`. This will post notifications as the motes step through the join process.

Figure 1 below shows five motes joined to a manager, as displayed with `sm`. Mote ID 1 is the AP (Access Point) mote. Make sure all the motes in your network show up on this list.



```
> login user

> sm
     MAC                   MoteId  State Nbrs Links Joins    Age StateTime
00-17-0D-00-00-38-0F-6C    1     Oper    4    20     1       0    0-00:13:44
00-17-0D-00-00-38-0C-86    2     Oper    2    10     1      10    0-00:13:23
00-17-0D-00-00-38-0C-91    3     Oper    2    10     1       7    0-00:13:05
00-17-0D-00-00-38-0C-96    4     Oper    4    13     1       7    0-00:12:41
00-17-0D-00-00-38-0C-99    5     Oper    3    14     1       5    0-00:12:35
00-17-0D-00-00-38-0C-89    6     Oper    1     9     1      25    0-00:01:28

Number of motes (max 33): Total 6, Live 6, Joining 0

> _
```

Figure 1 – CLI command `sm` showing the motes connected to the manager

33.6 Step 3: Configuring Motes for Specific Data Rates

Now that the network is up and running, we will need to configure the motes to generate and send data as if they were collecting it from sensors. We will specify the quantity, rate and size of the data we will send upstream (from mote to manager) through the network. We will use the PkGen utility available with SDK to accomplish this.

The packet generation utility PkGen allows for configuration of each mote in the network from the Manager itself. This means that the user does not have to physically go to individual motes to set their rate. Also, these configurations can be changed as desired, simply by entering new fields for the mote that needs the change. Since a SmartMesh network can operate as a heterogeneous network, each mote may be configured for its own data rate: you can have a combination of slow/medium/fast data rate motes, emulating the real world scenario of slow/medium/fast sensors.

Here are the steps to follow for configuring the motes using the PkGen utility:

For this section you will need to have the serialMux running

- Navigate to the `/win/` directory in the SmartMesh SDK, and launch PkGen by double-clicking on the Windows executable at `/win/PkGen.exe`
- It will give you a prompt where you choose between SmartMesh IP or WirelessHART. Pick the appropriate family and click **load**.
- If you are connecting to an IP device, just click connect through serialMux, when prompted. If this is not functional, make sure that the serialMux is installed properly in Step 1.
- If you are connecting to a WirelessHart device, enter the IP address of your manager when prompted. The default static IP address of the WirelessHART manager is 198.162.99.101. You can change it to any address desired. Please refer to the SmartMesh WirelessHART User's Guide for details.
- When you connect, the fields you entered should turn green and the list of motes in your network should populate the mote list section. Each mote in your network will have a series of fields in the table that you will enter to configure each mote as desired.
- The **mac** column gives the MAC Address of the mote connected to the network, which is to be configured. You may use this to determine the location of the mote in the field and then determine specific configuration.
- The next column, **num. pkgen**, is a counter of the packets sent to the manager from that mote at any given time once you start PkGen.
- The next column, **pk./sec**, is another counter that shows the average packets per second rate at which the mote is sending packets to the manager. It may take awhile for this to show up.
- The previous two columns can be reset by pressing the **clear** button. You will likely want to do this every time before you set new send parameters.
- The last column has three fields and the **set** button. The first field is for the number of packets total the mote will send. Depending on the duration of the deployment and the rate at which the mote is going to send data, enter the appropriate number here. You must send at least 1 packet, but there is no max. The second field is the interval between packets in milliseconds. (*e.g.* 1000 = 1 packet/ 1000ms or 1 packet per second). The final field is the size of the packet you want to send each time. (max 60 bytes with a 20 byte header).

⚠ Do not fill in the blank cells to be filled by the user with a "-". Leave them blank.

- Click on the **set** button on the last column to start the motes sending packets. (The **clear pkgen** button allows you to reset these fields, if desired.)

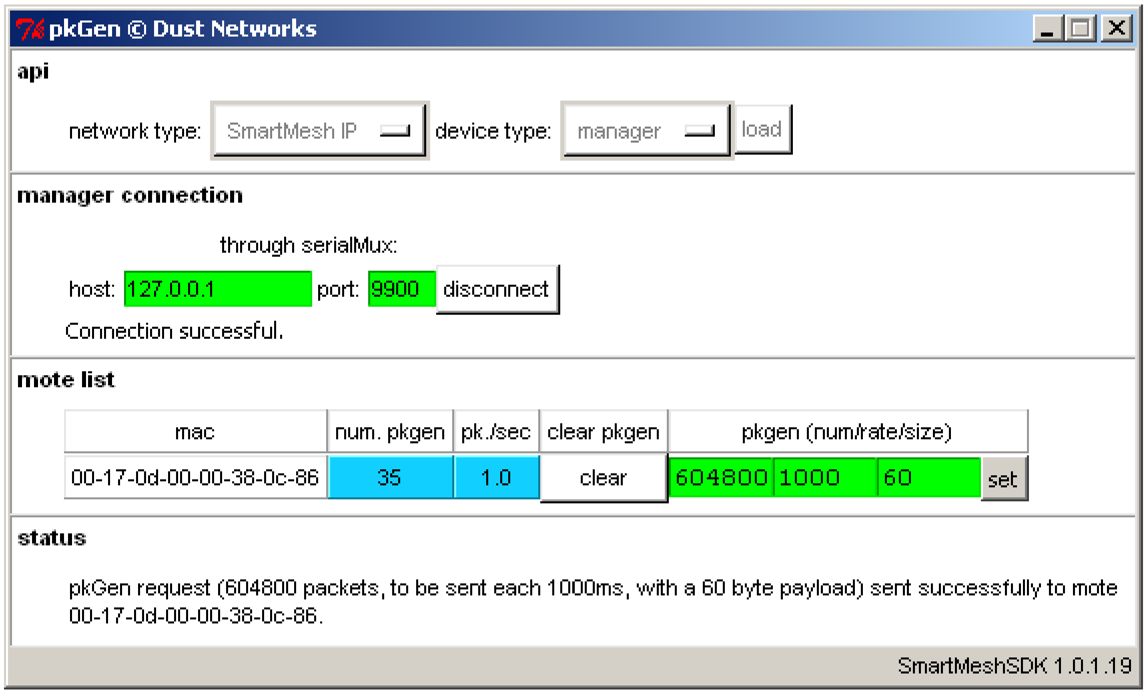
⚠ Wait between presses of the **set** button. It you hit **set** twice quickly, the cells may turn green erroneously indicating that they were updated, but no packets will be sent into the network.

- Repeat the above 2 steps for each of the motes in the network.

⚠ There is an upper limit to the total bandwidth available at the manager to receive packets - it is family-dependent, but ~25 packets/s. This limit should not be exceeded. In other words, the sum total of the data packets per second coming from all motes going to the manager should not exceed this limit. The data-rate at individual mote thus depends on the total number of motes in the network and their individual data rates. For example in a network with 5 mote, you should not exceed 5 packets per second on all the motes at once.

- Closing the script does not mean that the motes will stop sending packets. They will continue to do so until they have sent all packets. Reset the motes if you want them to stop them from sending data.

Figure 2 below shows an example of the PkGen screen shot. For example, If you wanted to send 1 packet per second at max size for a week, you would fill out the **num/rate/size** fields: **604800 1000 60**
Once you have set the parameters to your liking for each mote you should see the **num. pkgen** counter start counting. The **pk./sec** column will display the rate rounded to the nearest 0.1 packet per second. If the rate is too slow or not enough packets are sent (for it to calculate an average) it will display a rate of 0.0.


pkGen © Dust Networks

api

network type: SmartMesh IP device type: manager load

manager connection

through serialMux:

host: 127.0.0.1 port: 9900 disconnect

Connection successful.

mote list

mac	num. pkgen	pk./sec	clear pkgen	pkgen (num/rate/size)			
00-17-0d-00-00-38-0c-86	35	1.0	clear	604800	1000	60	set

status

pkGen request (604800 packets, to be sent each 1000ms, with a 60 byte payload) sent successfully to mote 00-17-0d-00-00-38-0c-86.

SmartMeshSDK 1.0.1.19

Figure 2 - PkGen screen shot showing configuration of a mote

33.7 Step 4: Gathering Statistics

Now that the network is sending data upstream, the next step is to gather network statistics that can be used to analyze the performance of the network in its current configuration. Every mote in a network periodically sends a packet which contains information about the network, packet success rate, neighbor information etc. These packets are called *health reports* and are typically sent every 15 minutes. A SmartMesh network has built in APIs that provides access to all the network statistics to the user, just as the data access.

You will take a snapshot of the statistics in one of two ways, depending on if you are using SmartMesh WirelessHART or SmartMesh IP

Step 4A: Gathering Statistics in an IP network

The SmartMesh IP manager maintains a minimal set of statistics for network reliability, path stability, and mote behavior. To get a complete picture of network health, the user must log all health report notifications. This topic is discussed in more detail in "Application Note: Monitoring SmartMesh IP Network Health."

Step 4B: Gathering Statistics with WirelessHART NetworkSnapshot

The SmartMesh WirelessHART manager contains a utility for collecting all manager logs - it is invoked by logging into the manager and using the command at the Linux prompt.

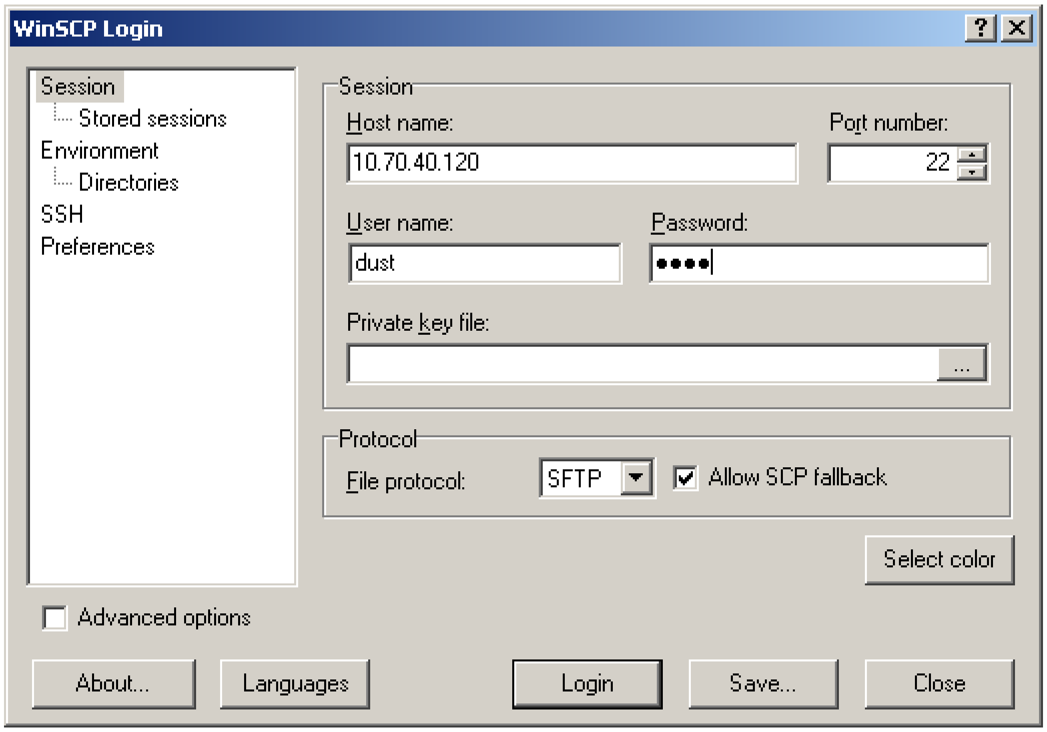
1. Log onto the manager using the dust login the way you would to connect to CLI.

2. Instead of using nwConsole you will use the command

```
/usr/bin/create-network-snapshot
```

This will take a snapshot of current network statistics and store it in `/tmp/snapshot/snapshot.tar.gz`

1. You can access this snapshot by connecting to the manager with WinSCP or another FTP client. You will connect to the same IP address you used to connect with pkgen and port 22. Navigate to `/tmp/snapshot` and transfer `snapshot.tar.gz` to your machine.


WinSCP Login

Session
Host name: 10.70.40.120
Port number: 22
User name: dust
Password: ●●●●
Private key file:

Protocol
File protocol: SFTP
☑ Allow SCP fallback

☐ Advanced options

Figure 3 - Transferring Network Snapshot data from the manager

⚠ Taking a new snapshot overwrites the old one so make sure to transfer the file off of the mote before taking the new one.

33.8 Step 5: Analyzing Data

Example tools to streamline the collection of snapshot logs in SmartMesh IP, and to analyze the logs produced by the snapshot features are still in development, but will be part of the SmartMesh SDK when available.

One goal of inspecting snapshot logs is to obtain data about path RSSI so that it can be compared to distance in order to build better networks. By finding which paths have stronger signal you will be better able to judge the optimal mote spacing.

33.8.1 Wireless Hart Snapshot Log

This log will be found in the .tar file you extracted from the manager. Once you have unzipped it the data you seek is found in the `nwconsoleOut.txt` file. This log, accordingly with its name, is in fact just the output of an internal querying of the console on the manager. It issues a variety of commands to gather information about the state of the network at the time of snapshot. The first commands, beginning with `show ver` give basic information about the version, status and settings of the manager in general.
`sm –a` will print the list of motes in the network.

Here is a segment of that output.

```
< sm -a
Current time: 07/24/12 09:15:03 ASN: 32492664
Elapsed time: 3 days, 18:15:30
MAC MoteId Age Jn UpTime Fr Nbrs Links State
00-17-0D-00-00-1B-1B-CD ap 1 1 3-18:15:23 6 10 106 Oper
00-17-0D-00-00-38-0C-86 2 16 2 19:02:46 2 2 11 Oper
00-17-0D-00-00-38-0C-89 3 9 2 1-00:03:00 2 7 17 Oper
```

The `get paths` command that follows will list every mote pairing and the number and direction of links on that path, as well as path quality. If no links exist it will be listed as unused and have a default path quality of 75.

```
< get paths
pathId: 65538
moteAMac: 00-17-0D-00-00-1B-1B-CD
moteBMac: 00-17-0D-00-00-38-0C-86
numLinks: 5
pathDirection: downstream
pathQuality: 90.07
pathId: 131077
moteAMac: 00-17-0D-00-00-38-0C-86
moteBMac: 00-17-0D-00-00-38-0C-96
numLinks: 0
pathDirection: unused
pathQuality: 75.00
```

Following this will be more commands with useful but less applicable information, but then there are the `show mote -a` commands that make up the bulk of the information you will want. These will show all the paths to neighbors for each mote as well as the RSSI and path quality for each of those paths.

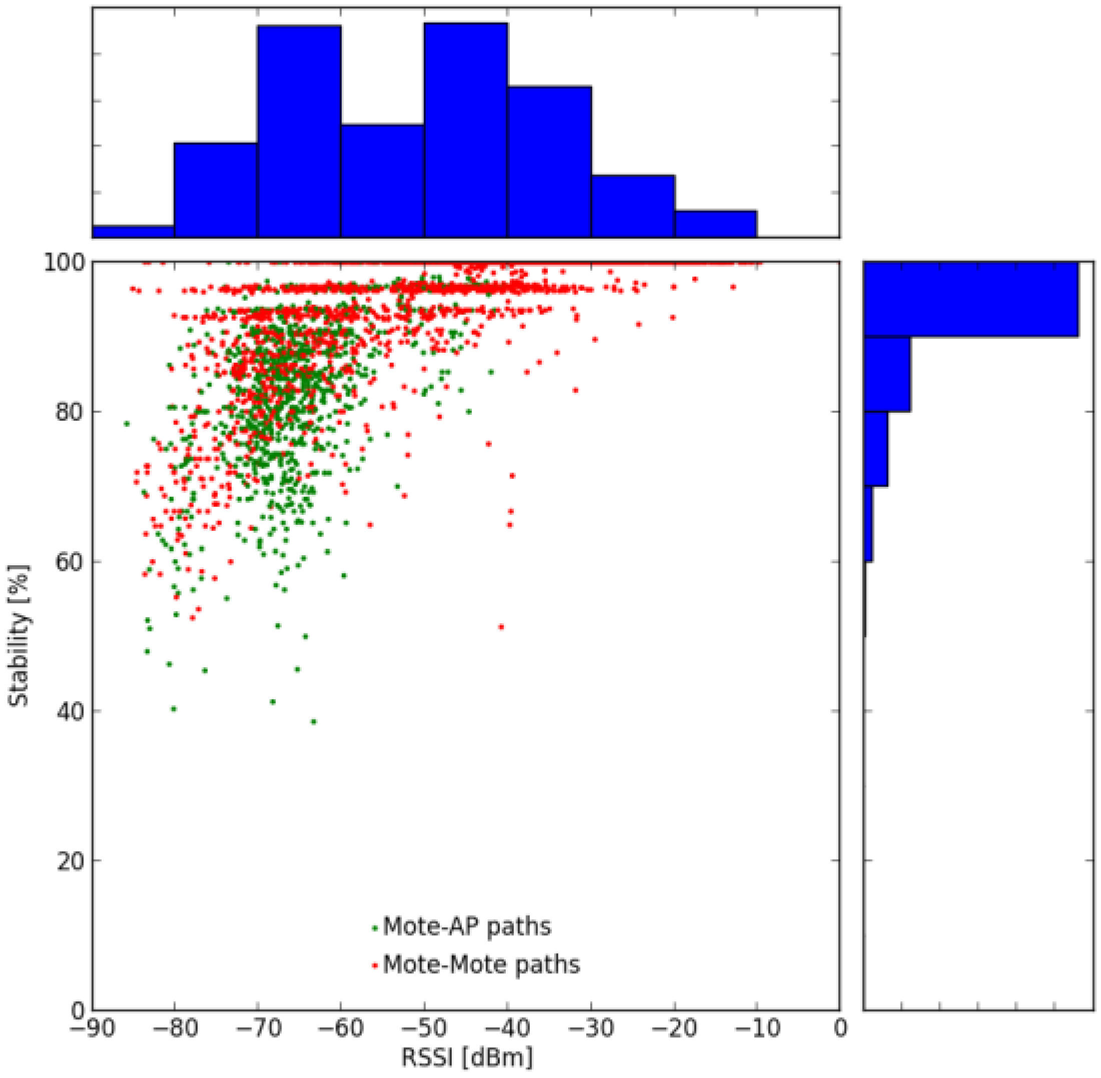
```
< show mote 1 -a
00-17-0D-00-00-1B-1B-CD 1 Oper SW: 3.0.2-0 HW: 37
Location is not supported
Number free TS: 757
Upstream hops: 0, latency: 0.000, TTL: 127
SourceRoute: Dist(Des): 0.0(10) Prim: 1 Sec:
Power Source: line
Advertisement Period: 20.000
Bandwidth:
Summary for AP: 2.5342
Neighbors: 10. Links: 106. Norm/bitmap 21/85 (max norm: 63)
Links per second: 19.824219 (unlimited)
Frame: 0. Neighbors: 10. Parents: 0. Links rx:87, tx:1.
Broadcast links
0. 1. 0: rtdb
0.993.10: rjb
<- #2 Links 3/3/3 RSSI: -52 Q: 0.90
<- #3 Links 4/4/4 RSSI: -29 Q: 0.94
<- #4 Links 8/8/8 RSSI: -50 Q: 0.83
<- #5 Links 5/5/5 RSSI: -51 Q: 0.86
<- #6 Links 3/3/3 RSSI: -48 Q: 0.90
<- #7 Links 33/33/33 RSSI: -41 Q: 0.86
<- #8 Links 16/16/16 RSSI: -43 Q: 0.90
<- #9 Links 6/6/6 RSSI: -45 Q: 0.87
<- #10 Links 4/4/4 RSSI: -41 Q: 0.90
<- #11 Links 3/3/3 RSSI: -41 Q: 0.95
```

Following this section the log will be populated with many tables of stats; daily stats for motes, paths and the network, as well as lifetime stats for the same.

33.8.2 Log File Interpretation and Analysis

The goal is to generate a waterfall plot that lists distance versus RSSI or path stability. The knee of this plot gives the user a good idea of what range is appropriate in the deployment environment. This range may be used for future deployments as opposed to the default of 50 m that was used in the original deployment.

A sample plot is shown below.



These plots can be generated by:

- Extracting the RSSI (or path quality) values for each path from the log file
- Calculating the distance between each mote pairing
- Plotting each RSSI (or path quality) value against the corresponding distance

33.8.3 Neighbor Stability Analysis

The number of good neighbors (strong path stability) a mote has is crucial to forming resilient networks. Gathering this data from the log files can help find weak links in the network mesh. This feedback would be particularly useful to someone deploying a network as motes with 3 or more good neighbors have a high probability of successfully joining the network. Those with less than 3 good neighbors will likely need more neighbors to be placed nearby.

Other network heath and performance may be obtained as described in the relevant family "Application Note: How to Evaluate Network and Device Performance."

34 Application Note: What is Packet ID and why do I Need it?

34.1 Scope

This document describes the utility of the Packet ID (bit 1) in the fields flags found in the Mote API header in both Wireless HART and SmartMesh IP Mote products. The first section of this document briefly describes the Packet ID. The second section describes how there are two Packet IDs being used across the Mote-to-Sensor Processor interface, each of which being toggled and tracked independently. The third section has some useful tips on how your code should use Packet ID and the related Sync Bit.

34.2 What is Packet ID?

Imagine we have a motorized bucket filler, and we have a communication interface where we instruct our device to:

1. Move to the right
2. Check that there is a bucket
3. Release one bucketful of water
4. Repeat

We could write a microcontroller application that tells our bucket filler to do this, and it would usually work. Imagine that we want this interface to be more robust to communication errors, so we implement a *call-response* protocol. For each command from the client (here the microcontroller), we generate a response from the server (here the bucket filler). Our transactions start to look more like this:

1. Microcontroller: Move to the right
2. Bucket filler: I moved to the right
3. Microcontroller: Check that there is a bucket
4. Bucket filler: There is a bucket
5. Microcontroller: Release one bucketful of water
6. Bucket filler: I released one bucketful of water
7. Repeat

Now we have a more robust interface, since the client gets confirmation that its message succeeded. But we can anticipate further problems when the message does not go through. If the microcontroller sends the command “Move to the right” and receives no response, it could be due to the bucket filler not getting the command, or the microcontroller not getting the response - in the former case the client needs to repeat the command, whereas in the latter it should move on to the next command to avoid confusing the server. What we need is an identifier that will allow us to tell the difference - we can do this by sending a counter along with the command:

1. Client: Move to the right (ID = 0)
2. Server: I moved to the right (ID = 0)
3. Client: Check that there is a bucket (ID = 1) - the command does not go through
4. No response received, so the client repeats the last command
5. Client: Check that there is a bucket (ID = 1)
6. Server: There is a bucket (ID = 1)
7. Client: Release one bucketful of water (ID = 2)
8. Server: I released one bucketful of water (ID = 2) - the response does not go through
9. No response received, so the client repeats the last command
10. Client: Release one bucketful of water (ID = 2)
11. Server: knows that it already filled the bucket in response to command (ID = 2), so it drops the command and replies "I released one bucketful of water (ID = 2)"

Because there was an identifier on the command, the receiver of the command can distinguish between a retry and a new command. The bucket filler knew that it didn't need to fill the bucket again, because it did not receive a different ID on the command. For *reliable* interfaces that don't advance to the next command until the current command is acknowledged, a 1-bit counter can be used for ID, since we only need to distinguish between a command and the command that follows. Command 1 follows command 0, and command 0 follows command 1. This is how the 1-bit Packet ID field works - by toggling the Packet ID bit between 0 and 1, the receiver can distinguish between a retry and a new command. Most Mote commands don't have a bad consequence like filling a bucket twice or moving two steps to the right instead of one, but it is still a major improvement to the reliability of an interface.

34.3 There are Two Packet IDs

In a device containing a Mote and a Sensor Processor, there are two independent communications streams - one where the Mote is server and the Sensor Processor is client, and another where the Sensor Processor is server, and the Mote is client. Each stream has its own Packet ID. Every time the Sensor Processor sends a command to the Mote, it must toggle its Packet ID. The Mote will check the Packet ID on every command received, comparing it to the previous Packet ID received. If the Packet ID is different than the stored value, then the command is processed and this Packet ID is stored. If the Packet ID is a repeat, the command is not processed and the Mote will reply with the same reply it sent on the previous command (the cached response).

Since the Mote can send commands (Notifications) to the Sensor Processor at any time, there is a second Packet ID for that communication. Every time the Mote sends a command, it will toggle its Packet ID. The Sensor Processor must check the Packer ID, comparing it to the previous Packet ID received. If the Packet ID is different than the stored value, then the Notification should be processed, and this Packet ID should be stored. If the Packet ID is a repeat, the Notification should be dropped and the microcontroller should send the previous cached response.

34.4 The Sync Bit

If the Sensor Processor application resets or for any reason, it no longer knows which Packet ID to use or to expect. The Sync bit allows the client to reset the Packet ID to a known state. The Sensor Processor should set the Sync Bit on the first packet it sends to the Mote - this tells the Mote that this is a new packet, regardless of the Packet ID. The Sensor Processor must toggle the Packet ID from that point forward. When the Sensor Processor receives the first packet after it resets, it should accept any Packet ID and expect the Packet ID to toggle for further packets. In SmartMesh IP, should the Mote reset, it will always send a boot event with the Sync Bit set. In SmartMesh WirelessHART, the Sync Bit will not be set, but the Sensor Processor can treat each boot event as unique, so it can accept any Packet ID from the Mote and expect the Packet ID to toggle for further packets.

34.5 Ignoring Packet ID

In order to simplify a Sensor Processor implementation to just get it working, the Sensor Processor can effectively ignore Packet ID by processing every command on the receive side assuming it is new. On the transmit side, it should set the Sync Bit on every message, ensuring that the Mote will process every packet as a new command. You will lose some of the robustness of the reliable call-response interface by doing this, so this is not the recommended behavior for anything but a prototype.

35 Application Note: Monitoring Background Noise

35.1 Introduction

SmartMesh IP networks automatically collect and publish data to monitor the in-band noise throughout the network. Motes publish Health Reports containing this data every 15 minutes. For every channel used by the mote, it collects the average background RSSI, and the number of transmit attempts and failures. This data gives the application a virtual distributed spectrum analyzer that looks at the effects of interference throughout the network. If there are certain channels that are underperforming, the user could choose to set a blacklist for the network and reboot, or could use the information to locate an interferer in order to remove it from the environment, if possible.

35.2 Feature Details

To use this feature, both the motes and manager must be of the following (or later) versions:

- SmartMesh IP Mote Stack 1.4.1
- SmartMesh VManager 1.0.1 or SmartMesh IP Embedded Manager 1.4.1

Motes automatically take measurements and send health reports containing the measurement data and the manager automatically publishes the data through notifications. The user does not need to take any action to activate the feature. However, it is the user's responsibility to act upon the results as the manager does not change any network behavior based on background noise measurements.

There are two components of the feature: background RSSI measurement and channel stability.

35.2.1 Background RSSI Measurement

Because of the unpredictability of RF signals, motes are provisioned with more bandwidth than will typically be used. As such, motes will have receive links during which they wake up but their neighbor has no packet in its queue to transmit on the link. From the receiver's perspective, we call this event an *idle listen*. It is possible that the mote overhears a transmission that it locks on to but for which it isn't the destination, in this case the mote will drop the packet but it doesn't count as an idle listen. At the end of every idle listen, the mote takes a quick low-power measurement of the RSSI on the channel it was listening to. There should be no 802.15.4 traffic from the network on the channel at this time. Each time the mote makes a measurement on an idle listen, the RSSI is averaged with other measurements on this channel. If there is background interference (*e.g.* from another SmartMesh network, a WiFi network, or other 2.4 GHz band devices) near the mote on a particular channel, we would expect the mote to measure higher RSSI values on that channel.

35.2.2 Channel Stability

Every time a mote sends a MAC-layer unicast message, it expects to get an ACK from the recipient. If the mote doesn't hear an ACK after transmitting a packet, the mote counts this transmission as a failure and will retry the packet transmission at a later time. To track channel stability, the mote keeps a count of the number of times it transmits on each channel and the number of failures on that channel. From these two counts, we can calculate the channel stability. For example, if the mote transmitted 10 times on channel 5 and failed 2 times, the channel stability is 80%. Similar to before, if there is background interference near the mote on a particular channel, we would expect the mote to have a lower channel stability on that channel.

35.3 Health Report Packet Format

The Channel-Specific Health Reports contain 15 channels worth of data. Each packet is classified as an Extended Health Report with a netExtHealthNotif ID.

Type	Description
INT8U	TLV tag (Always 0x01 for RSSI Reports)
INT8U	TLV length (length of payload below)
RSSI_REPORT_T[15]	RSSI reports for 15 channels (structure below)

Where each RSSI report consists of:

Type	Description
INT8S	Average idle listen RSSI on the channel
INT16U	Number of unicast attempts on the channel
INT16U	Number of missed acks on the channel

Here is an example of the output printed from a VManager trace where each channel is broken out on its own line:L_TRACE RX ie=32:1e mesh=4022 ttl=126 asn=19661 gr=1 src=83 rt=g tr=u:req:0; sec=s ofs=23 nc=27 ie=fe:00 IP=7e77 UDP=f7 sP=f0b0 dP=f0b0; cmd=HR
len=77hr=01:4b:a5:00:10:00:01:a6:00:15:00:02:aa:00:15:00:00:a6:00:0d:00:01:ad:00:11:00:01:a7:00:19:00:01:ac:00:0f:00:04:a

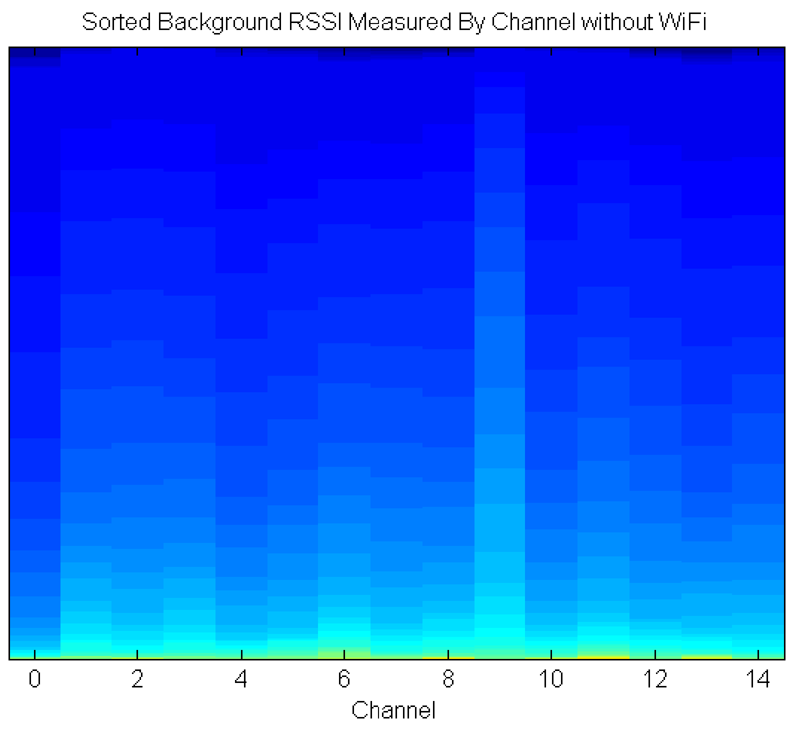
The highest background noise measured by this mote is on channel 4 where the average RSSI was -82 dBm (from the hex value of `0xad`). To calculate the channel stability for channel 1, *s* = 1-2/21 = 90.5%.

35.4 Interpreting the Results

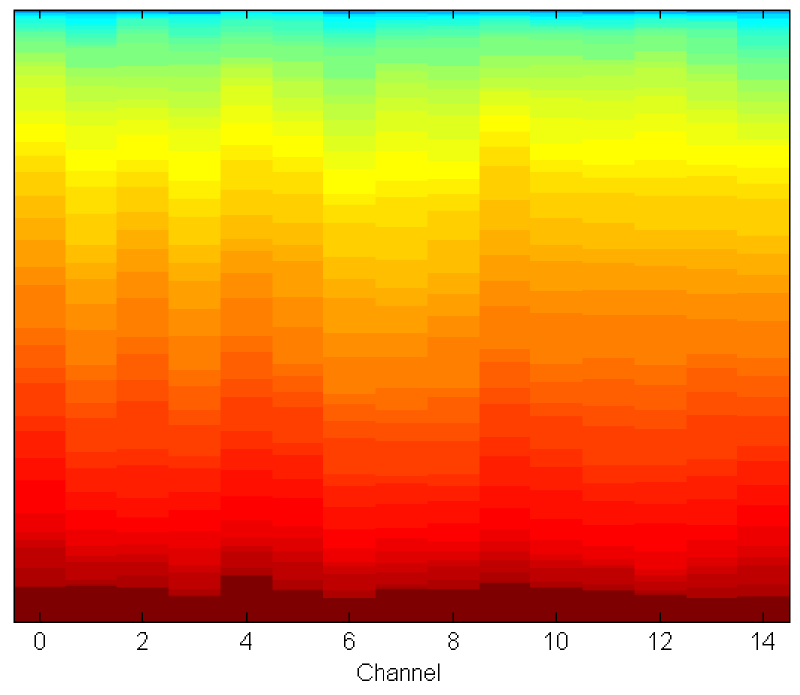
Fundamentally, we don't care about background noise unless it is causing packets to fail in the wireless network. To this end, channel stability is a more valuable metric than the background RSSI measurement. However, background RSSI can provide some clues to where the interference is located. For example, suppose there are two motes A and B with A transmitting upstream to B. If A reports bad channel stability on channel 5, but a low background RSSI of -90 dBm, it is unlikely that this interference is causing the packet failures. In that case, check the background RSSI for channel 5 reported by B. A high background RSSI, say -60 dBm, reported by B would explain why A is having trouble succeeding on this channel to its parent.

35.4.1 Results without Interference

We set up a network with 250 devices, 64 of which were configured to send background noise health reports. We let the network run for approximately three hours and gathered about 12 health reports from each of the 64 reporting motes. For this network, we were interested in looking at the global properties of the noise as opposed to trying to locate any source of interference, so we lumped all the health report data into a single set of distributions. For each channel, we looked at all the background RSSI values reported by motes for that channel, and the channel stability for that channel. The results are summarized below where each column represents one channel's worth of sorted data.



Sorted Channel Stability without WiFi



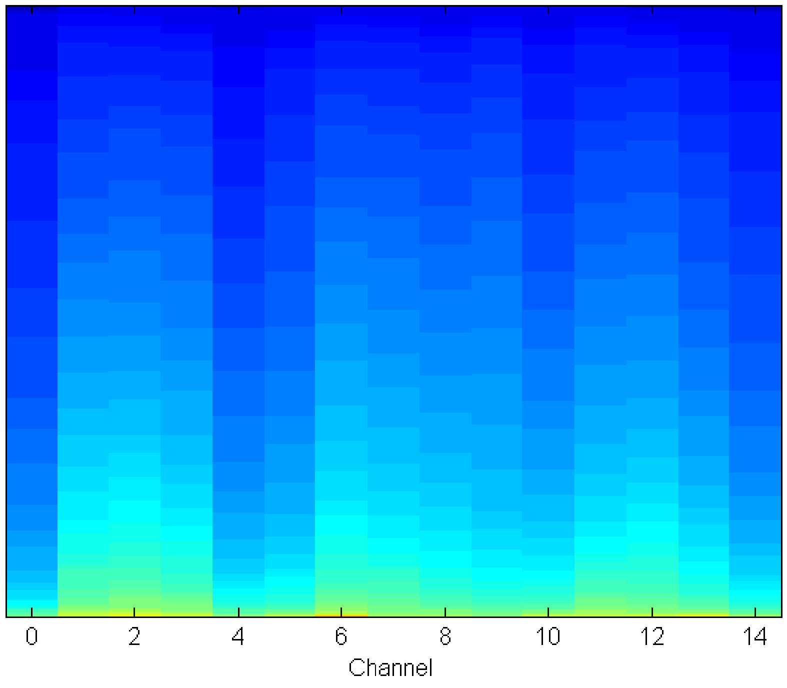
For the background RSSI, solid blue represents an average measurement of -100 dBm over the 15 minutes of idle listen events. These solid blues are sorted to the top of the plot and aren't prominent. The solid green color represents an average of -67 dBm, and the lack of much green, yellow, or red on the plot suggests that the vast majority of the time, average RSSI is measured below -67 dBm. Furthermore, while there is a small amount of noise measured on channel 9 (apparently coming from a WiFi network in a neighboring office), there are no significant peaks of interference on any channel.

A similar structure is used for the channel stability plot. Here solid red represents 100% success on that channel over 15 minutes and solid blue represents 0% success, i.e. all attempts on that channel failed in this interval. The midpoint, solid green, is 50% stability. Again, the stability is relatively uniform across all channels.

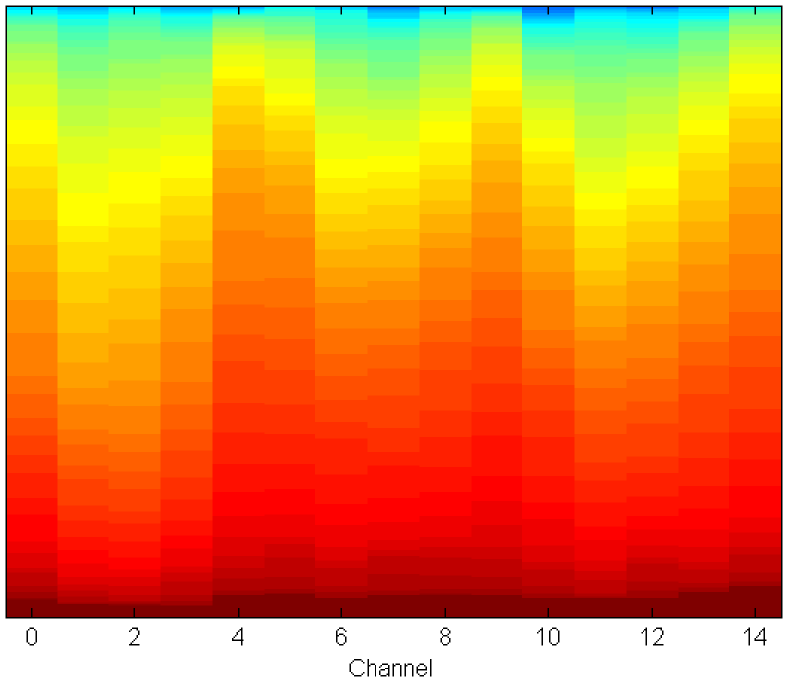
35.4.2 Results with WiFi Interference

For the second part of the test, we activated the 2.4 GHz band of the WiFi for the building. The two plots below use the same color scales as before.

Sorted Background RSSI Measured By Channel with WiFi



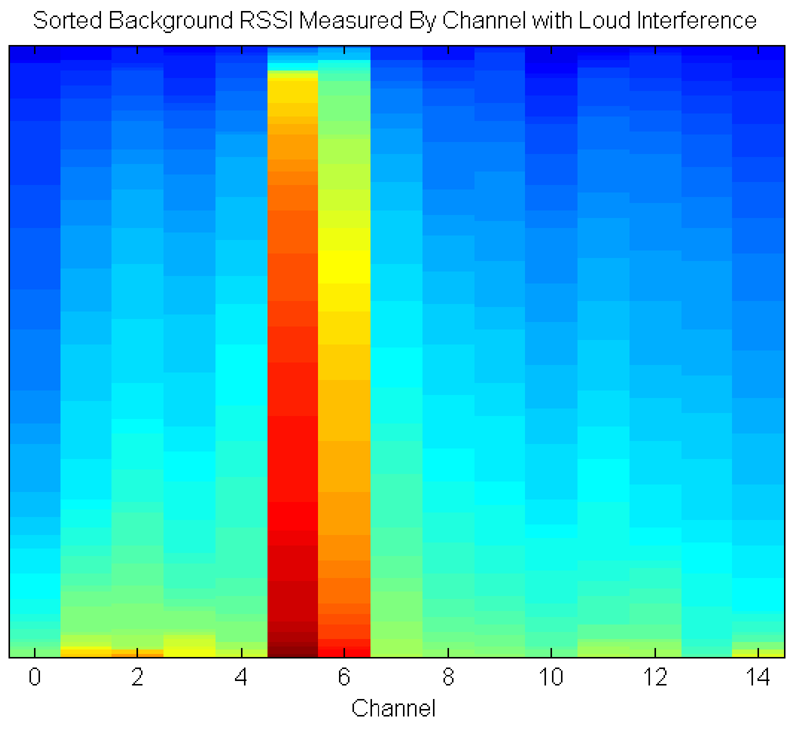
Sorted Channel Stability with WiFi

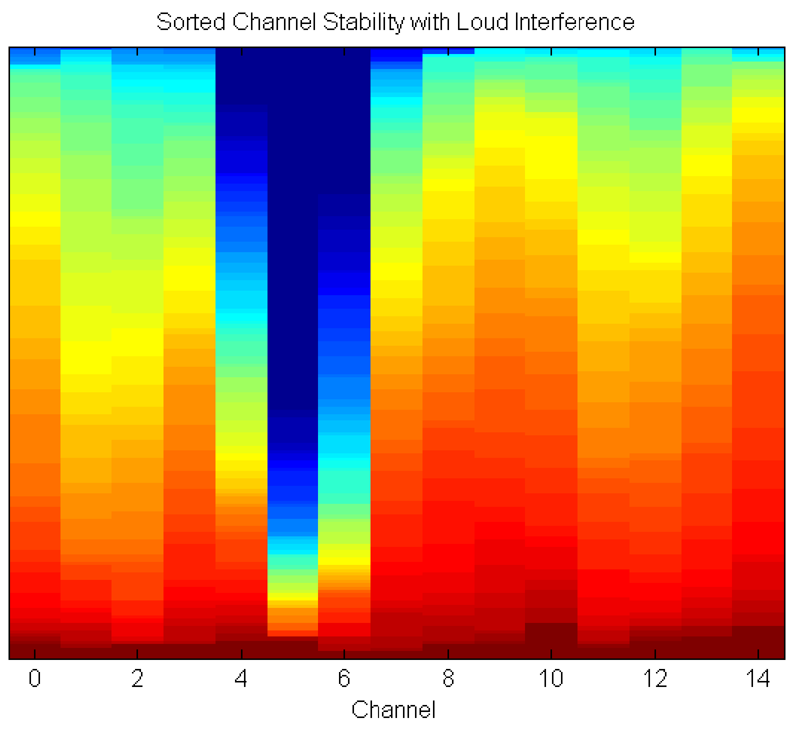


On the background RSSI plot, the WiFi interference is visible as three separate bands, each about three channels wide. This makes intuitive sense as WiFi channels are about three times as broad as 802.15.4e channels. The small peaks in the background RSSI plot do indeed correspond to small decreases in stability on the same channels suggesting that when the WiFi is active it can cause failed transmissions in our network. The good news is that overall the impact is minimal as the overall network stability dropped only 4% with the addition of WiFi. SmartMesh networks are configured to automatically hop around interference, and this is an example where exactly this is happening. Empirically, if motes are measuring background RSSI on a subset of the channels but stability is largely unaffected, no mitigating action should be taken.

35.4.3 Results with Constant Narrow-Band Interference

For the third part of this test, we continuously transmitted loud GSM-encoded interference on a 7 MHz bandwidth. The transmit power of this system was the same as our WiFi access point, but unlike WiFi, this interference was not duty cycled. The effects are much more severe on two channels as shown in the plots below.





Channel 5 is severely impacted and the adjacent channels also have significant stability decreases. This interference is so severe that practically any transmission in the network that uses channel 5 will fail.

35.4.4 Blacklisting Based on Results

If the interference is restricted to a small number of channels but is pervasive throughout the network, consider setting a blacklist using the *PUT /network/config* command (VManager) or the *setNetworkConfig* command (Embedded Manager). There are a few caveats to blacklisting. Firstly, setting a blacklist requires a network reset. Secondly, once channels are blacklisted, motes will no longer be able monitor background RSSI or measure channel stability on the blacklisted channels so it will be difficult to determine if the interference pattern has changed. Thirdly, the blacklist is network-wide, so this solution does not help if there are different interferers in different parts of the network. After a blacklist has been set for a network, compare the stability before and after. If the stability has improved, then the performance of the network will improve: latency and average mote energy consumption will both decrease. In our sample results above, we would not recommend any blacklisting in the WiFi-only case but would recommend blacklisting channels 4, 5, and 6 in the narrow-band interference case. Blacklisting can be particularly effective to help latency and reliability if any channel has less than 33% stability.

If the interference covers the whole band and the whole network, the only solution is to increase the transmit power of the devices in the network. This can be done by adding a power amplifier to the radio circuitry to bring the output power to as high as is allowed by local regulations. Increasing the link budget with a receive-side amplifier will not help as the noise will be amplified as much as the desired signal and the SNR will not improve. If this is not possible, consider either increasing the provisioning in the network to allow more retries per packet generated or installing repeaters between pairs of devices that have bad path stability.

If the interference is localized, the location of the affected motes can be used to pinpoint the source of the interference. In some situations it may be possible to remove the interferer or reduce its output power to improve network performance.

We have had cases of very loud out-of-band interferers (*e.g.* WiMax) causing stability problems at the edge of the band. The channel stability measurements can be used to discover these sources even though direct in-band measurements of RSSI may not show any energy.