Sensor-Rocks: A framework to improve software management for sensor networks

Timothy Telfer[*], Sameer Tilak[†], Philip Papadopoulos[†], and Luca Clementi[†]
[*]Monash University, Melbourne, Victoria, Australia
[†]University of California, San Diego, La Jolla, CA, USA
Email: tdtel1@student.monash.edu, {stilak, ppapadopoulos, lclementi}@ucsd.edu

***Abstract*—**
Software Operations and Management (O&M), i.e., defining, configuring, and updating thousands of software components within a conventional data center is a well-understood issue [12]. Existing frameworks like the Rocks toolkit [7], [11] have revolutionized the way system engineers define, deploy and manage large-scale compute clusters, storage servers, and visualization facilities. Unfortunately, the state-of-the-art approach in sensor network software O&M relies on system administrators to manage the operating system and overall system configuration of a system as a golden image. This image is manually configured according to site and individual project requirements. In the manual approach, this golden image is copied onto corresponding sensors typically using Over The Air Programming (OTAP) [8], [9], [10]. The fundamental problem with any golden image approach is that the methodology often used to build this software master is either unspecified or ad hoc. In either case, only the author of the golden image can change its contents, add new functionality, update existing components or completely rebuild (rebase) the image when the underlying OS changes. The manual approach does not scale, since ensuring consistency of installation, and accuracy of configuration and updates of thousands of software components is laborious and error-prone. One could dramatically reduce both time and effort while improving system reliability if techniques used to deploy scalable computing systems (e.g., data centers) were extended to the challenging world of sensor networks. We propose to develop Sensor-Rocks by adapting and extending the Rocks toolkit. We believe that Sensor-Rocks will revolutionize the world of sensor networks by reducing the administrative overhead of defining the software environment on individual sensors and network of sensors from days/weeks to a few hours. The resulting faster, reliable deployment of the software infrastructure will fundamentally improve the reproducibility, which is a key for gathering good-quality data.

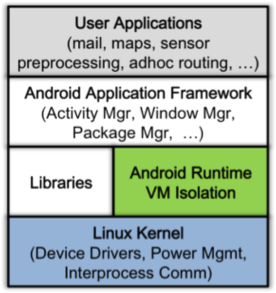
I. Introduction

The Android platform has significant share of the mobile phone market and it is run on a broad range of devices including embedded platforms (e.g., GumStix), tablets, and netbooks. We believe that Android will play a significant role in the next generation of environmental monitoring applications since it allows one to leverage the significant technological investment made in producing what has become commodity embedded systems. Therefore, we choose to use the Android platform for the proposed research. In Section II we describe the current software O&M practice in Android world and describe how the proposed approach will allow one to rapidly and reliably build complete Android environments (firmware flashes). In Section III we motivate the need for automating software O&M on the Android platform, give an overview of the Rocks toolkit, and then in Section III-B we describe our approach. In Section III-C we describe the future work and in Section IV we present conclusions.

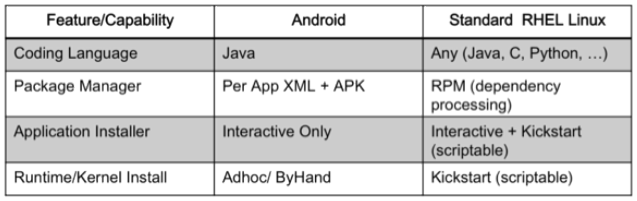
II. Background

Android is an operating environment for portable, low-power devices that delivers a unified framework for developing mobile applications system. It consists of a customized Linux-based kernel, which provides the essential support for the physical hardware and attached devices. Applications are insulated from the specific hardware and isolated from each other using the Android application framework, a set of core libraries, and a virtual machine abstraction (Dalvik VM) for application isolation (See Figure 1(a)). Android applications are authored in Java and can be rapidly developed using a freely-available software development kit (SDK). Most developers interact only with the SDK and never need to change/modify the lower layers (this is NOT true for sensors, we must add device drivers and other libraries). Since nearly all Android devices are destined for human-in-the-loop interaction, software delivery, installation, and configuration is almost always done interactively with a person dragging and clicking on a touch screen. This interaction modality is a significant and practical barrier for deploying Android on 100s to 1000s of sensors with no screens. Nearly every Android-based device (like a cell phone) has its complete software stack provisioned in two steps. First, the device manufacturer creates a system image with pre-loaded applications and the full support stack and then flashes this image (firmware) onto the device. This first step scales to thousands of identical hardware devices. Second, users customize their devices by adding additional applications through an online Market or direct download. Both parts – the manufacturer's image and user-installed applications can also be updated over the air.

Because there exists only a relatively small number of physical device variants (October 2011 estimates about 100 distinct Android-based cell phone devices), little attention has been paid to developing a top-to-bottom framework for creating a device's base (flash) image. Instead, software engineers handcraft images specific for their proprietary devices. This person-


User Applications
(mail, maps, sensor preprocessing, adhoc routing, …)
Android Application Framework
(Activity Mgr, Window Mgr, Package Mgr, …)
Libraries
Android Runtime
VM Isolation
Linux Kernel
(Device Drivers, Power Mgmt, Interprocess Comm)

(a) Android application and kernel stack



Feature/Capability	Android	Standard RHEL Linux
Coding Language	Java	Any (Java, C, Python, …)
Package Manager	Per App XML + APK	RPM (dependency processing)
Application Installer	Interactive Only	Interactive + Kickstart (scriptable)
Runtime/Kernel Install	Adhoc/ ByHand	Kickstart (scriptable)

(b) Some key system definition differences between Android and Full Linux

Fig. 1. Android software stack and its comparison with Linux

intensive methodology is quite understandable because of the very small number of physical variants, but it also generally leads to stagnant firmware. In this way, the state of Android firmware definition is not unlike that of Linux from 15 years ago, before the development of integrated releases represented by Slackware and RedHat. Like today's Android devices, those early adopters of Linux downloaded all necessary components, compiled and then assembled a working, complete OS in a hand-crafted process that could take days, weeks or even months. Today, not only is it easy to install a Linux-based OS on a wide variety of hardware, but automated installers can detect specific hardware while supporting user-defined custom stacks to create a fully-operational computing system with no interactive steps in under an hour. Linux could evolve to its state today of supporting millions of hardware variations only when the basic system definition became modular. Android must (and we predict eventually will) follow a similar path to system modularity as hardware platforms proliferate. It is important to explore improved mechanisms for Android system stack definition for three key reasons: 1) inexpensive sensor devices have no interactive screens and must therefore be configured only using a flashed-system image; 2) scaling to 100s or 1000s of sensors means that we must be able to handle hardware heterogeneity of individual sensors without the time consuming process of building a highly-customized, independent image for each variant; and 3) we want better reproducibility of the basic software configuration so that we can easily adjust to the rapid changes of the Android environment and reap the benefits of new capabilities. In short, to support scalable sensor deployments we must have a significantly more nimble definition environment.

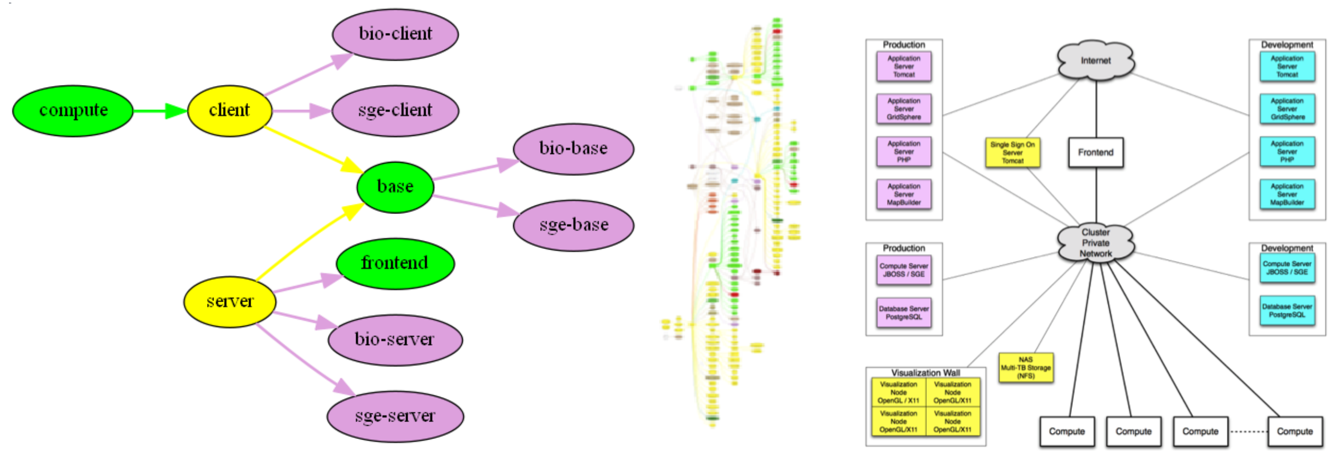
Figure 1(b) shows some other important differences between Android and full Linux. Android can be thought of as a very specialized Linux environment where applications can be written only in Java, have specific modes in which they can interact with the system, and operate on just a few hundred hardware variants. However, we generally expect the offerings in Android to dramatically expand as devices become more capable and Android based platforms like Gumstix [4], Pandaboard [6], BeagleBoard [1] (without specific vendor/carrier lock-ins) proliferate. We believe that in the not too distant future (within next 3-5 years), a complete Android environment will have the capabilities and flexibility of a commodity Linux node of perhaps 10 years ago. This means that our intended approach to modular system footprint construction will become essential as system complexity grows. Such a construction technique will require dependency processing of packages and scriptable (non-interactive) configuration as first steps. One can accelerate the development of some of these now *tried and true* techniques used on fully-capable Linux systems to support our goals of many sensors (hardware heterogeneity) and variants of sensor configuration (functional heterogeneity) in the rapidly changing Android software environment (a new major release occurs every 6-9 months with updates on a monthly basis). Package definition and dependency can be achieved by adopting well-known package formats (e.g. RPM or Debian's dpkg) in combination with definitions of Android-specific software repositories or sets of packages. Automated dependency resolution can be handled via YUM (RPM-compatible) or APT (dpkg compatible) so that all necessary software prerequisites can be added simply by choosing a specific application. However, automated full-systems configuration is much more challenging for two reasons: 1) no scriptable system definition framework like Redhat's Kickstart exists today for Android ; 2) Unlike installed Linux systems, common tools for modifying configuration files via scripting languages are not part of the installed Android environment. Linux can automatically define and configure itself (this is what happens when you install from a DVD), but Android does not have the same closed form. Some other environment must be used to create the completely configured firmware. In essence, we will have to cross-compile an Android configuration using a Linux host.

III. Automated Software Operations and Management (O&M)

In this section, we give an overview of the Rocks toolkit and then describe our approach for automating software O&M for sensor networks.

A. *Rocks Background:*

Rocks is a software toolkit that solves the computing cluster definition, deployment and management problem [7], [11]. It has reduced the time from raw hardware to a working system within the data centers from days/weeks to a few hours. Rocks scales out to 1000+ node environments within the resource-rich environment of a data center. The toolkit treats a complete


compute
client
bio-client
sge-client
base
bio-base
sge-base
frontend
server
bio-server
sge-server
Production
Application Server Tomcat
Application Server GridSphere
Application Server PHP
Application Server MapBuilder
Internet
Development
Single Sign On Server Tomcat
Frontend
Cluster Private Network
Compute Server JBOSS / SGE
Database Server PostgreSQL
Visualization Wall
Visualization Node OpenGL / X11
NAS Multi-TB Storage (NFS)
Compute

(a) A Portion and a Complete Rocks Config. Graph. Server and compute appliances are defined by expanding instructions contained within each connected node. Different colors indicate different Rolls.

(b) A Rocks-based sample data center architecture

Fig. 2. Rocks-based approach for automating software O&M in a data center environment.

software footprint on any machine as a set of software packages and configuration that together form a Rocks Appliance. Appliances can share packages and configuration and the approach takes advantage of the many similarities among login nodes, compute nodes, web servers, storage servers, and visualization walls.

Rocks splits the definition of any appliance into two fundamental pieces: the distribution, which is the complete set of software packages (including OS Packages) and the Rocks configuration graph [11]. Software installation on a given appliance is performed by starting at an appliance-defined entry point in the configuration graph and then traversing this specific subgraph. The nodes in the graph itemized both the required software packages and subsequent configuration needed to make software functional. This information is compiled to create a description that is given to a native installer like Redhat's Kickstart or Solaris' Jumpstart. The installer (now operating in a completely automatic mode) finishes the last steps of formatting disk drives, installing packages, and following the configuration instructions generated by Rocks. A key observation is that the graph is *source code* for a complete set of appliances and that this in-depth and prescriptive information can be used in novel ways, including traversing the graph to support online updates and partial traversals to support reconfigurations or additions of software and also for version tracking and management.

Rocks is quite extensible by others and fundamentally removes the unspecified or ad hoc process of golden image creation. To achieve extensibility, Rocks defines a structured set of programmable, replaceable and extensible components called Rolls. Major software systems are structured as rolls. The base operating system, the core tools of Rocks, and kernel rolls are required. However, numerous optional addons like web-services, database engines, task schedulers, high-performance computing tools, Hadoop support, Xen virtual machine hosting, Android authoring environment, and more, have been created. Rolls enable experts to capture exactly how complex software should be configured [11] and transfer that knowledge in the form of executable code (the Rocks graph). Rolls provide both the architecture and mechanisms that enable the end-user to incrementally and programmatically modify the graph description for all appliance types. The toolkit enables non-cluster experts to deploy and run clusters in a matter of hours instead of days or weeks.

Figure 2(a) is a representation of a sample core Rocks subgraph, which describes the configuration of appliances in a Rocks cluster. This configuration graph is composed of nearly two hundred vertex files and a graph XML file from each included Roll. Each vertex file specifies the complete software package and configuration information for a specific function. The graph file specifies the directed graph edges that connect the vertices to each other. Using this XML representation of a configuration graph, complete software configuration for any cluster node type can be produced by traversing the configuration graph from the appropriate entry point vertex. For example, the configuration graph in Figure 4 has two vertices named ”compute” and ”frontend”, which are the entry points for building compute nodes and frontends. Rolls further decompose this configuration from a single graph of hundreds of nodes, into multiple sub-graphs, which are assembled at installation time into what was previously a single monolithic framework. Rocks has been able to configure cluster complexes with a data center that are defined as integrated systems of computing, web-servers, database engines, authentication servers, and other heterogeneous functionality (nearly a dozen distinct appliances) working together. Figure 2(b) is an example of one such data center that was built using the Rocks toolkit for the Community Cyberinfrastructure for Advanced Marine Microbial Ecology Research and Analysis project. It is the back-end system that powers the Portal at: (http://camera.calit2.net).

Today, Rocks supports systems in the 1000s of nodes including cluster complexes that are defined as integrated systems of computing, web-servers, database engines, authentication servers, and other heterogeneous functionality working together. Similarly the process of building field-deployed sensor appliances will also involve building dozens of appliances and configuring and installing numerous software components. It provides an excellent basis for developing scalable, extensible, and modular software infrastructure to automate software O&M for the sensor networks.

B. Proposed Approach

We propose to adapt to sensors and sensor networks a well-understood methodology in the space of computing clusters that already has demonstrated scalability to 1000s of nodes and transparent support for heterogeneous hardware. This entails extending and adapting the existing Rocks graph-based framework to describe the configuration of all node types (appliances) that make up a complete sensor network. This version of the Rocks software (*Sensor-Rocks*) will enable faster and more reliable deployment of the sensor cyberinfrastructure (CI) thereby fundamentally improving system reproducibility. Fundamental to our approach is the fact that it does not use *golden* images as a management tool. Instead its processing programmatically builds a functional and fully-configured system from smaller, sharable and reusable components. Different sensor types and roles within a sensor network are directly analogous to appliances in a Rocks cluster. Although we believe that the methodologies and concepts are equally applicable in the sensor network domain, there are many open issues that need to be explored.

We utilized CyanogenMod as the initial base operating system for deployment on the sensor devices [2]. CyanogenMod is a custom operating system based on Android, providing significant enhancements many of which useful in the domain of sensor networks. These include (1) Older device support (2) Greater battery efficiency (3) leaner software footprint (4) Nightly builds (5) CPU under/over-clocking, and (6) Wifi/Bluetooth/USB tethering.

1) Sensor-Rocks method: We borrow the concept of configuration graphs (along with conditions and constrains) and rolls. Following the Rocks methodology, the entire set of possible appliances within the sensor network is represented as a profile graph. A profile graph (ref. Figure 3(a)) consists of a number of interconnecting rolls, each of which represent their own software systems. Within the graph, individual rolls are represented as rectangles, and each has a unique color. The rolls are further described by a set of nodes, each of which contains information describing the packages and configuration of a specific function of that roll. The nodes are represented on the graph as ovals (ref. Figure 3(a)), and have a matching color to the roll they belong to. The profile graph applies conditional edges to allow certain paths to be included or excluded based on the value of attributes (such as hardware type). As shown in Figure 3(b), to allow for dependencies between the nodes, an ordering graph describes any dependency constraints between them.

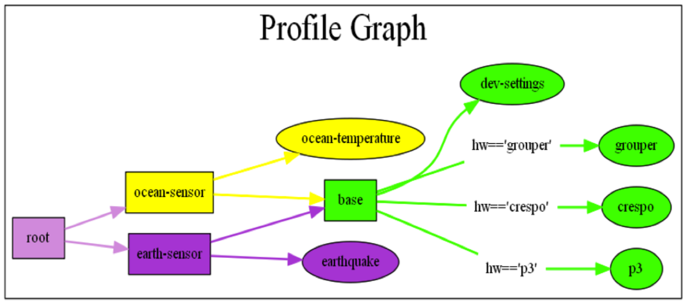
To build binary image specified in this profile graph, the software iterates through a set of directories representing each roll in the profile. The roll directories each contain packages, scripts, and a set of XML files to describe the roll subgraph and its nodes. A roll XML file provides meta information on that roll, such as development details and representation data. A subdirectory of node XML files describe the packages and configuration settings for each node contained in the roll. Graph XML files are then used to define the structure of the nodes within this roll, how it is attached to the overall profile graph, and any ordering constraints and attributes. Figure 4(a) shows the proposed workflow involved in building and installing an appliance configuration. Blocks colored blue represent tasks performed by the user, green represents tasks performed by sensor-rocks and orange represents tasks performed by the device. Blocks with text preceded by (*) represent tasks that are not yet present or automated in the current framework. The current workflow assumes the packages are precompiled, the user will enter the device type attribute and manually push the rom to the device.

In the example profile graph, Figure 3(a), three rolls are included, along with a root node to keep everything connected. The base roll contains the base operating system (Cyanogenmod) that will be run on the given hardware device. Figure 3(a) shows three device nodes attached to the base roll, each representing the Cyanogenmod binary for that specific device. Codenames have been used in this example, as per the Android and Cyanogenmod standards. Asus Nexus 7 is referred to as ”grouper”, Samsung Nexus S is referred to as ”crespo” and Samsung Galaxy Tab 10.1 is referred to as ”p3”. These devices are included as they were used during testing of the software. The dev-settings roll includes post-scripting to apply settings to the devices, useful for development purposes. The ocean-sensor roll and its subsequent node represent the necessary packages and settings for an ocean-sensor device, that are included on top of the base roll. The earth-sensor roll and its node contain packages and settings necessary to create a sensor device to measure for earthquakes.

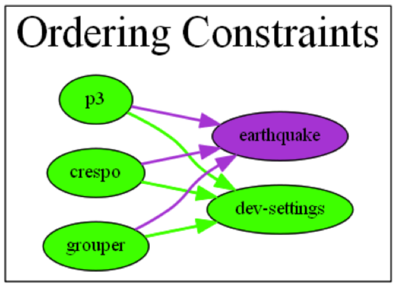
2) Sensor-Rock compiling distributions: Using the sensor network's profile graph, a specific appliance can be built by traversing the graph from a given starting roll, and applying any defined attributes to the conditional edges. This process results in a subgraph of nodes, representing the contents of a unique appliance's image. Using the information contained in each node from this subgraph, an ordered list of packages and configuration scripts are produced. An example of this is described in the ”Building a Device Using Sensor-Rocks” section below.

The Rocks approach uses a script in Kickstart file format [5] to automate the installation process. Sensor-Rocks implements similar functionality using Edify scripting [3], which can be used to instruct the Android installation process to mount file systems, change permissions, extract packages, and many other tasks. Each node can contain its own Edify scripts, which are combined into one large installer-script, while the appliance distribution is created. This combined script is placed into a zipped archive along with the contents of all the node's packages. The resulting zip can be loaded onto the device and installed to produce the desired sensor device.

Sensor-Rocks also includes an important piece of functionality from the Rocks methodology which is refered to as ”post-scripts”. These are scripts which are packaged as part of the



(a) Sensor-Rocks Profile Graph



(b) Ordering Constraints

Fig. 3. Describing profile graphs and declaring ordering constraints

rom, but instead of being a part of the installation process, they are run after the device has booted. Both Rocks and Sensor-Rocks implement these by allowing the user to write shell scripts, and scheduling them to run after the device has booted. Post-scripts are split up into two main types: ”Run-Once” and ”Run-Always”. Run-Once post scripts will be scheduled to run after the device has first booted, and are then discarded. These can be useful for applying device settings, such as enabling and disabling communication methods. The Run-Always scripts are not discarded after first-boot, but instead will be run everytime the device boots. These can be useful for performing tasks, such as connecting to a specific network.

3) Sensor-Rocks module details: The Sensor-Rocks framework utilizes a modular design, to allow interchangeability between major components in the system, for a more flexible tool set. It consists of three major modules: Package Manager, Roll Manager and Device Manager.

The Package Manager: Its purpose is to handle the source code repository control for all packages included in the sensor-network. It is able to retrieve new source code when it is available, and compile it into package binaries, ready to be included in a roll. For this iteration of the Sensor-Rocks Framework, the package manager is incomplete, and instead the rolls utilize pre-compiled binaries.

The Roll Manager: Its responsibilities include handling the xml files describing the sensor-network's rolls, create visual graphs representing the sensor-networks rolls and nodes, and creating ROM images ready to be loaded onto sensor appliances. This is the focus of this iteration of the Sensor-Rocks framework.

The Device Manager: Its goal is to manage the devices already deployed in the sensor network. It will contain a database of all devices deployed and relevant data about them. It will also handle the initial installation process of a created ROM onto the device, removing the need for the developer to manually perform this task. Future work will focus on creating and optimizing an update process for an already deployed sensor-network, enabling updated updates (i.e., source code or profile graph structure) to be automatically pushed and installed onto sensor devices, without the need for manual intervention.

4) Building a Device Using Sensor-Rocks: Sensor-rocks is controlled using a command line interface on a Unix terminal. The following is an example process performed by a user of Sensor-Rocks wanting to deploy an appliance using the profile graph in Figure 3(a).

Figure 4(b) shows the set of commands involved in building a device using Sensor-Rocks and installing the binary image onto it. The first command creates a subgraph of the overall profile graph, representing the nodes to be included in the appliance configuration. The subgraph starts at the requested earth-sensor roll, and includes all nodes it passes by following each directed path. If a node has a condition on the attribute ”hw”, that node will only be included if the condition requires the value ”grouper”. In this case, the crespo, and p3 nodes are not included, but the grouper node is. The final subgraph created will contain the nodes: earth-sensor, earthquake, base, dev-settings and grouper. This is all compiled into a ROM, and is labelled ”rom.zip” in the output directory.

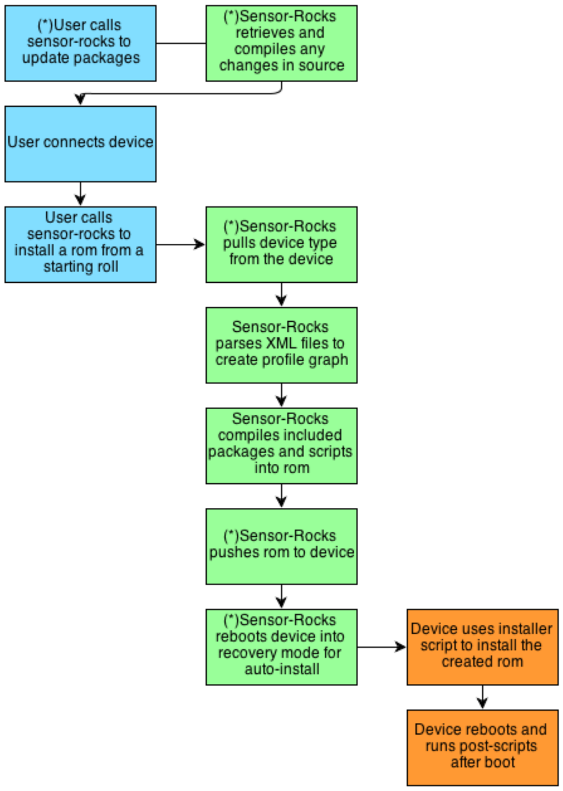
The two ”adb” steps are used to push the ROM onto the SD card of the device, and then reboot the device into recovery mode, so that rom can be installed. The user must then select to ”Install ROM from SD card” and then locate the ROM. After that, the device will install the ROM, and then reboot into it. These two steps will later be automated as part of the Device Manager module, following the workflow seen in Figure 4(a).

C. Future Work

The next stage for the Sensor-Rocks project is to perform a scalability analysis. We will quantify the time taken in the process of deploying sensor networks with Sensor-Rocks and compare this with the time taken to deploy the same network using the previous golden image approach.

Future research and development of the Sensor-Rocks framework will aim to further automate the tasks required in deploying and managing a sensor network. One area of focus will be pulling properties from a connected device to automate the filling of configuration attributes. Another major area of focus will be automating the process of pushing and installing updated ROMs onto the devices within an already deployed sensor-network.

Increasing ease of use is another future aim of the project. One of the tasks involved in this will be creating helper tags throughout the node xml files, automating some of the post-scripting or install-scripting that the use would previously have


(*)User calls sensor-rocks to update packages
(*)Sensor-Rocks retrieves and compiles any changes in source
User connects device
User calls sensor-rocks to install a rom from a starting roll
(*)Sensor-Rocks pulls device type from the device
Sensor-Rocks parses XML files to create profile graph
Sensor-Rocks compiles included packages and scripts into rom
(*)Sensor-Rocks pushes rom to device
(*)Sensor-Rocks reboots device into recovery mode for auto-install
Device uses installer script to install the created rom
Device reboots and runs post-scripts after boot

(a) Sensor-Rocks Workflow.

1	sensor-rocks-cm.py compile-roll -r earth-sensor -a "'hw' : 'grouper'"
2	adb push <path-to-sensor-rolls>/out/rom.zip /sd-card/
3	adb reboot recovery

(b) Commands for building a device using Sensor-Rocks framework.

Fig. 4. Sensor-Rocks workflow and commands to build devices.

to create. One such example would be a helper tag for setting the network access point for the device, where previously the user would have to create a post-script to perform this activity, instead the helper tag would instruct Sensor-Rocks to create it automatically.

Another feature to be included that will increase ease of use, is an intuitive graphical user interface. The user will be able to select a starting roll and attributes, and be able to see the resulting sub-graph as he makes the changes. This will help the user ensure he is getting the correct configuration on the appliance. The graphical user interface will also display information and graphing of the current state of the deployed sensor network, allowing easy overview and management. The controls included within the interface will greatly decrease the learning curve required to operate the Sensor-Rocks Framework.

IV. Conclusion

Long-term deployments of sensor-based environmental observing systems need an efficient and scalable software Operations and Management (O&M) approach, and a power management approach that meets the functional, operational, financial, and policy requirements. In this position paper, we describe an integrated approach that utilizes, adapts, and extends the Rocks toolkit that has worked well in a resource-rich data center environment to the resource-constrained world of sensor networks. Sensor-Rocks framework will enable much faster and more reliable deployment of the sensor cyberinfrastructure (CI) thereby fundamentally improving system reproducibility. Its programatic approach, rather than a golden-image based approach, is modular and scalable. In this paper we also described the design and implementation details of the Sensor-Rocks framework. We have used it to successfully build a variety of images for multiple hardware platforms.

ACKNOWLEDGMENTS This work is supported by a grant from the Gordon and Betty Moore Foundation and NSF awards 1219504 and 1234983.

References

[1] Beagleboard. http://beagleboard.org/.
[2] cyanogenmod. http://www.cyanogenmod.org/.
[3] Edify script language. "http://forum.xda-developers.com/wiki/Edify_script_language.
[4] Gumstix. http://www.gumstix.com/.
[5] Kickstart file format. http://fedoraproject.org/wiki/Anaconda/Kickstart.
[6] Pandaboard. http://pandaboard.org/.
[7] G. Bruno, M. J. Katz, F. D. Sacerdoti, and P. M. Papadopoulos. Rolls: modifying a standard system installer to support user-customizable cluster frontend appliances. In *CLUSTER*, pages 421–430, 2004.
[8] J. W. Hui and D. E. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys*, 2004.
[9] P. Levis and D. Culler. The firecracker protocol. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, 2004.
[10] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and energy efficient data dissemination service for extreme scale wireless networks of embedded devices. *IEEE Transactions on Mobile Computing*, 6(7):777–789, 2007.
[11] P. M. Papadopoulos, M. J. Katz, and G. Bruno. Npaci rocks: tools and techniques for easily deploying manageable linux clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):707–725, 2003.
[12] S. Tilak and P. Papadopoulos. The case for a rigorous approach to automating software operations and management of large-scale sensor networks. In *Computer Communication Review*, volume 42 of *ACM SIGCOMM*, pages 58–61. ACM, 2012.