The Case for a Rigorous Approach to Automating Software Operations and Management of Large-scale Sensor Networks

Sameer Tilak
Calit2, UCSD
stilak@ucsd.edu

Philip Papadopoulos
SDSC, UCSD
ppapadopoulos@ucsd.edu

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

ABSTRACT

Software Operations and Management (O&M) i.e., installing, configuring, and updating thousands of software components within a conventional Data Center is a well-understood issue. Existing frameworks such as the Rocks toolkit [8] have revolutionized the way system administrators deploy and manage large-scale compute clusters, storage servers, and visualization facilities. However, existing tools like Rocks are designed for a *friendly* Data Center environment where stable power along with high-performance compute, storage, and networking is the norm. In contrast, sensor networks are embedded deeply within the harsh physical environment where node failures, node mobility and idiosyncrasies of wireless networks are the norm. In addition, device heterogeneity and resource-constrained nature (e.g., power, memory, CPU capability) of the sensor cyberinfrastructure (CI) are realities that must be addressed and reconciled. Although sensor CI must be more adaptable and more-rapidly reconfigurable than the data center equivalents, few if any of the existing software O&M tools and techniques have been adapted to the significantly more challenging environment of sensor networks. A more automated approach to software O&M would provide significant benefits to system builders, operators, and sensor network researchers. We argue that by starting with software O&M techniques developed for data centers, and then adapting and extending them to the world of resource-constrained sensor networks, we will be able to provide robust and scientifically reproducible mechanisms for defining the software footprint of individual sensors and networks of sensors. This paper describes the current golden-image based software O&M practice in Android world. We then propose an approach that adapts the Rocks toolkit to allow one to rapidly and reliably build complete Android environments (firmware flashes) at the individual sensor level and extend to a large networks of diverse sensors.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics—*management, reproducibility*

Keywords

Sensor networks, smartphone, Android, software operations and management

1. MOTIVATION AND PROBLEM DEFINITION

Large-scale sensor networks enable scientists and engineers to instrument the world at an unprecedented scale, providing deep measurement on important issues such as climate variability and change, ocean acidification, desertification, water sustainability, and impact of large-scale hazards on civil infrastructure. These systems range in size from mid-scale (on the order of hundreds of sensors as deployed by the Global Lake Ecological Observatory Network [3] and the Coral Reef Environmental Observatory Network [2]) to large-scale (on the order of thousands of sensors as being designed by the Ocean Observatories Initiative [6] and the National Ecological Observatory Network [5]). With the advent of personalized sensing (sensors integrated with consumer electronics and carried by people) it is possible to track user experiences and support applications such as personalized medicine to tens of thousands of users [9].

Both the environmental observing systems and personalized and mobile sensing environments are complex enterprises consisting of two major components, (1) Field-deployed sensor networks (embedded cyberinfrastructure) connected to (2) a Back-end Data Center comprised of high-performance computing, storage, and visualization facilities. Given the scale and complexity of these systems, they face a critical challenge in software Operations and Management (O&M), i.e., installing, configuring, and updating thousands of software components on a variety of platforms ranging from embedded cyberinfrastructure to data centers spanning multiple administrative domains and countries.

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) [10, 11, 12]. 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 argue the need to develop a clear and programmatic mechanism for reproducibility of sensor network cyberinfrastructure. It will transform the world of field-deployed sensor networks by reducing the software O&M overhead from days and weeks to a few hours. The resulting faster, reliable deployment of the software infrastructure will fundamentally improve the reproducibility, which is a key for conducting good-quality science. The Android platform has significant market share in mobile phone market and it runs 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 personalized mobile sensing environments. In addition, our direct experience with multiple observing systems indicates that Android platform will play a key role in the next generation of environmental observing systems since it allows one to leverage the significant technological investment made in producing what has become commodity embedded systems. We now 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).

2. 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 2). 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 (e.g. cell phones, tablets, and set-top boxes), 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 hundreds to thousands 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 exist only a relatively small number of physical device variants (October 2011 estimates about hundred

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 engineer’s hand-craft images specific for their proprietary devices. This person-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 and there exist only hand-build techniques today for this step; 2) scaling to hundreds or thousands 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.

Table 1 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 base platforms like Gumstix [4], Pandaboard [7], 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

Figure 1: Android application and kernel stack

can like 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. This second observation is that Linux can be used to 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.

3. PROPOSED APPROACH

An automated software O&M proposed approach must result in images that can then be flashed onto devices for actual deployment. The key difference is that our goal is to construct these images through a programmed, reproducible, automated process instead of relying upon hand-crafted *golden images* for each combination of sensor type, sensor hardware, sensor role in the network, and version of Android. This allows us to have very good reproducibility of deployed systems and removes significant uncertainty about how fielded system is really defined. We now describe how a Rocks-based approach can be used to mitigate this challenge. Rocks is a software toolkit that solves the computing cluster definition, deployment and management problem [8, 13]. 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 enables non-cluster experts to deploy and run clusters in a matter of hours instead of days or weeks. The toolkit treats a complete 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. Fundamen-

tal to Rocks is 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.

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 [8, 13]. 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. Rocks is quite extensible by others and fundamentally removes the unspecified or ad hoc process of golden image creation. Rolls provide both the architecture and mechanisms that enable the end-user to incrementally and programmatically modify the graph description for all appliance types. By adopting Rocks' graph approach to system configuration and developing the appropriate package and configuration mechanics, we can then practically define collections of applications and underlying kernel/device support into something we term an *Android/Rocks appliance*. We also plan to adapt the concept of Rocks rolls to define major subsystems (e.g. adhoc routing, specific physical sensors, novel power management) that can be shared by multiple sensor appliances. Rolls are in some sense *super packages* but can also express the multi-tiered configuration of a cluster (for full Linux) or novel sensor networks.

The process of building and running Android on a Gumstix Overo board is complex. At high-level it involves the following steps (1) Obtain the source code for the Android port of Gumstix (e.g., <https://github.com/gumdroid>) (2) Building the kernel (3) Building the file system (3) installing driver to support the PowerVR SGX graphics hardware (4) bundling the file system (5) creating a bootable SD card (6) Installing the boot files (7) Installing additional drivers if any. The aforementioned manual approach to build a large-scale sensor network with heterogeneous devices is laborious and error-prone.

As a first step, we have already created a Rocks Roll that

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

Table 1: Android software stack and its comparison with Linux

enables a user to deploy a Android authoring *workstation* for full Linux systems. The roll can be rapidly updated when changes to any of the underlying software components change. If a newer version of Android has newer subsystem requirements, these can be expressed too. We have successfully compiled an Android image for the Gumstix Overo computer-on-module on a virtual machine running the Roll. Once the roll is created any user wishing to use it can install it in a matter of minutes. Our preliminary experiments show that running on a virtual machine, a clean install of Android authoring workstation using the Rocks roll was performed in less than 10 minutes.

We plan to adapt the techniques pioneered in Rocks to define the software environment of the Android sensor itself. We can already reliably reproduce the authoring environment; we want to reliably reproduce the software footprint of the sensor itself. This will allow us to configure software components/services (data acquisition, time synchronization, localization, routing, etc.) in a scalable and reproducible manner. In addition, we believe that the lessons learned from adapting Rocks to the Android environment will be directly applicable in adapting it other embedded operating environments.

4. CONCLUSION

Software Operations and Management (O&M), namely, ensuring consistency of installation, and accuracy of configuration and updates of thousands of software components is a critical issue in large-scale sensor networks. The current manual approach is laborious and error-prone. In this paper we use the Android platform as a case study since we believe that it will play a significant role in large-scale sensor networks such as environmental observing systems and personalized mobile environments.

We argue that it is important to explore improved mechanisms for Android system stack definition to ensure that we can configure and manage hundreds or thousands of heterogeneous inexpensive sensors only using a flashed-system image with little to no human interaction. By guaranteeing reproducibility of the basic software configuration we can easily adjust to the rapid changes of the Android environment and reap the benefits of new capabilities. If one does not create a clear and programmatic mechanism for generating software footprint on Android platforms, the hardware and capabilities of Android itself will far outpace our ability as software providers to create stable and flexible sensor infrastructure. To that end, we proposed an approach that utilizes and adapts the Rocks toolkit from the friendly, resource rich data center environment to the challenging, resource-constrained world of field-deployed sensor networks.

As a first step, we have already created a Rocks Roll that enables a user to deploy a Android authoring *workstation* for full Linux systems. The next step is to adapt techniques

in Rocks to define the software environment of the Android sensor itself.

5. ACKNOWLEDGEMENTS

We would like to thank Geoff Pascoe for his help with the implementation of the Rocks Roll that enables a user to deploy Android authoring *workstation* for full Linux systems. This work was partially supported by NSF grants OCI-1219504 and OCI-0627026.

6. REFERENCES

- [1] Beagleboard. <http://beagleboard.org/>.
- [2] Creon: The coral reef environmental observatory network. <http://www.coralreefeon.org/>.
- [3] Gleon: The global lake ecological observatory network. <http://www.gleon.org/>.
- [4] Gumstix. <http://www.gumstix.com/>.
- [5] Neon: The national ecological observatory network. <http://neoninc.org/>.
- [6] Ooi: Ocean observatories initiative. <http://oceanobservatories.org/>.
- [7] Pandaboard. <http://pandaboard.org/>.
- [8] 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.
- [9] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, R. A. Peterson, H. Lu, X. Zheng, M. Musolesi, K. Fodor, and G.-S. Ahn. The rise of people-centric sensing. *IEEE Internet Computing*, 12:12–21, 2008.
- [10] J. W. Hui and D. E. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys*, pages 81–94, 2004.
- [11] P. Levis and D. Culler. The firecracker protocol. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM.
- [12] 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*, 6:777–789, 2007.
- [13] 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.