A Software Framework for Integrated Sensor Network Applications

Raju Pandey and Joel Koshy
Department of Computer Science
University of California, Davis
Davis, California, 95616
(pandey|koshy)@cs.ucdavis.edu

***Abstract*— Sensor networks have received wide attention in recent years for their revolutionary impact in numerous fields. To harness their full potential, researchers are beginning to build end-to-end solutions that integrate heterogeneous sensor deployments with traditional networks. While such efforts will bring true value to the use of sensor networks, there are several challenges that need to be kept in perspective. This paper highlights some of the key challenges, and presents a system software methodology to meet the difficult demands of development, deployment and long term management of sensor network systems.**

I. Introduction

Wireless sensor networks (WSNs) are emerging as an important technology that allows tight integration of the physical world with a computing system infrastructure. These ad hoc, self-organizing, distributed systems are composed of a vast number of sensor nodes deeply embedded in physical environments, that cooperate to monitor and/or measure various phenomena, and enable a wide spectrum of applications such as habitat monitoring and health care.

A growing trend in real world applications is to integrate heterogeneous WSNs with traditional networks [1]. The objectives of this approach are to not only facilitate remote management and data collection (i.e., a strictly hierarchical architecture), but also to partition the application into local processing and remote processing components (a more vertically integrated architecture) that cooperate to result in truly large-scale and eventually planetary-scale applications. Diversity at each level is an inherent characteristic of such designs. As a result, the complexities of long term management are multiplied across the functional tiers of the network.

WSN applications are built over *software infrastructure* (operating systems, middleware, virtual machines (VM), etc.) that (i) interfaces the applications with low level devices, (ii) provides communication mechanisms for cooperation among devices, and (iii) manages device and network-wide resources. The process of building WSN applications has typically involved writing applications for a specific platform and operating system. There has been little effort in developing general software development methodologies that can be used for systematically modeling, developing, building and deploying WSN software infrastructure and applications. As a result, development, deployment and management of WSN applications have been extremely difficult, as developers have had to deal with these nuances by hand. This paper strives to develop a broad understanding of the roles that different software development methodologies can play in WSN application development. We argue that the software infrastructure design should not be driven solely by application functionality or device characteristics, but also by software engineering concerns. Ignoring these concerns will make real world applications for integrated WSNs harder to write, deploy and maintain.

A. Lessons from Software Engineering

Over the last fifty years, we have learned two key lessons in software engineering: (i) Application developers should focus only on the application and not the idiosyncrasies of hardware, system platform, or system infrastructure; and (ii) All software systems (applications, operating systems, middleware, etc.) are bound to change over their lifetime — to add capabilities, to fix bugs, and to patch up security holes.

Platform and infrastructure independence: In traditional systems, applications encounter platform diversity in several forms, including hardware, OS, and middleware. Platform independence argues for a software model in which developers focus on implementing the application logic. Platform-specific nuances are hidden in the implementation of abstractions that application writers use in their code. Separation of application and platform code promotes reuse both among applications — by factoring code into a shared system layer — and across platforms — by facilitating application porting.

WSN applications will deploy platforms that will exhibit much more complexity and heterogeneity, which arises due to the domain-specific nature of the applications and deployment models. Deployment models tend to be resource-driven, with highly optimized task-specific devices, each designed to meet specific sensing, processing, data fusion or routing needs. Indeed, real world deployments are increasingly hierarchical, organized into multiple tiers with *sensor nodes*, *gateways*, and *base stations*. Sensor nodes collect and process locally sensed data, and cooperate with neighboring nodes. Gateway nodes aggregate data, set up and maintain distributed infrastructure, and provide a collaborative point of control for a group of nodes. Base stations act as anchor points, and bridge the WSN to larger distributed infrastructure (such as the Internet, and scientific or corporate databases). Sensor nodes, gateways and base stations differ significantly in their architecture (8 bit, 16 bit, 32 bit, etc.), memory architecture, ISA, resources (e.g., SRAM), communication bandwidth, and power requirements. Heterogeneity among WSN devices will also be induced by long term deployment models: in several applications, devices will be deployed or replaced in phases over several years using succeeding generations of devices. Thus, at any given time, an application may deploy several generations of devices.

The challenge for WSN application writers lies in developing applications that can run on a wide variety of devices and can be easily evolved to run on a new set of devices when necessary. While there has been significant effort in supporting interoperability and portability through standardization and abstraction (e.g., APIs, VMs and middleware, etc.), many of these solutions cannot be directly applied, as they assume functional homogeneity among all hosts. Further, in a resource-constrained environment, conventional interoperability solutions such as VMs and middleware cannot be directly deployed, as they require significant amounts of computing, memory and energy resources.

What is needed is a software development methodology that will: (i) enable platform-independent development; (ii) scale software infrastructure and applications to run on resource-constrained devices (e.g., Mica and Telos) as well as more powerful devices (e.g., Stargate and XYZ), and (iii) enable applications to exploit specific capabilities and resources of each device.

Long term management of applications: 60–70% of software cost is incurred in maintaining software after it has been deployed [2]. About 50% of this effort is perfective, 21% corrective, 25% adaptive, and 4% preventive [3]. These statistics underscore the highly dynamic nature of software artifacts. While the nature and frequency of changes may differ in the WSN domain, both application and software infrastructure updates after deployment are inevitable.

Intrinsic support for software evolution is still an open issue in traditional systems. The problem is that most software systems (especially their cores) are not designed with long term software management and evolution as a fundamental concern. Consequently, a variety of workarounds such as wrappers and binary patches around the fixed core, have been used to support software management and evolution. This typically results in software that is bloated, fragile, and bug-ridden. To complicate things further, all major applications have their own form of application management and update solutions. Clearly, these solutions will not work in the WSN domain. Given the inaccessibility of sensing devices due to large scale deployment in physical settings that may be impossible to reach, it is critical to anticipate strategies for software evolution in the early stages of system design. Further, the energy cost of binary distribution must also be considered. *What is needed, is a software infrastructure with intrinsic support for long term software management, and that allows seamless co-evolution of WSN applications with software infrastructure to ensure that it remains viable for a long period of time.* The software evolution mechanism should: (i) support both application and software infrastructure updates; (ii) reduce the energy cost of distributing updates; (iii) operate with low latency; and (iv) function securely and reliably.

These requirements are central to the principles that we present in this paper. As researchers rush to build and deploy WSN applications, we should be mindful that a careless approach to software infrastructure design will result in short-lived applications that are bound to a single platform. The key challenge is, how do we build a software infrastructure that supports both platform independence and long term management of applications? In Section II, we describe a framework which we believe answers both of these concerns. Section III gives an example of a software infrastructure built upon these principles, and Section IV concludes.

II. Design Principles for Sensor Network Software Infrastructures

The SENSES project [1] at UC Davis is developing a WSN software infrastructure for the development, deployment and long term management of WSN applications. The infrastructure addresses the challenges of

[1] `http://senses.cs.ucdavis.edu`

resource constraints, platform diversity and WSN system evolution, through *scalable* component-based VM abstractions that can be adapted post-deployment through an incremental evolution technique. Functional layering and software components are the key ingredients. The granularity of the ingredients dictates the level of flexibility that is afforded. Software synthesis, software scaling, and insitu adaptation provide the flexibility needed to build a broad array of software infrastructures. The framework can be visualized as a factory method that manufactures precisely tailored software infrastructure for specific applications.

A. *Layered architecture and software reification*

The traditional top-down program development approach conditions application writers to think about software in terms of layers. Service layers are the result of functional partitioning of the software environment such that all included functional processes perform at the same level of abstraction. Each layer provides a well-defined set of services to the layers above it, and services themselves may be implemented by a set of layers. Layering provides a conceptual organization of software, and realizes the principle of separation of concerns.

The layering paradigm is hard to achieve in WSN applications due to the severe resource constraints of end devices. We would like to preserve the advantages of a layered architecture and at the same time, meet the limitations of the underlying hardware. This can be achieved by constructing a *reified view* from the *logical view* of the layered architecture. The application programmer sees the logical view when developing software components, so that his task is unhindered by peculiarities and limitations of the target platform. The reified view is an application-specific *projection* of the logical view on a specific device. Reification of the layered architecture may cause actual layer boundaries to shift to optimize the architecture for performance, but this is transparent to the application programmer. Reification also applies to software components created by application and system programmers. While the programmer may view them as resident in a certain layer, their implementations may be specialized to cut across layers. There are two principal aspects to software reification: (i) software synthesis, and (ii) software scaling (Figure 1).

Software synthesis

Tailoring applications to fit onto devices, or automated software synthesis from a set of software components, is a well-established technique in programming embedded systems. Synthesis is necessary because there is often a tendency to make layered architectures too general. This results in an unnecessarily rich service model with

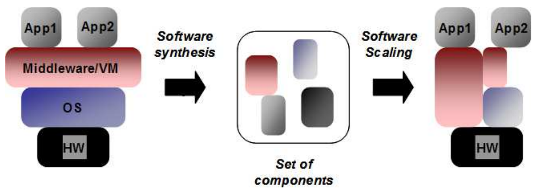

Fig. 1. Software synthesis and scaling.

a very narrow application layer (Figure 1). Cost and size constraints of devices dictate that WSN devices will contain processors with limited memory resources, and it is not possible to load an entire suite of services. Semantic analysis is used to statically analyze the application. This helps determine the composition of a minimal, resource-efficient infrastructure which is then instantiated and loaded onto the target. Current embedded software infrastructures are usually statically configured by the programmer (e.g., SUN's KVM). Although such systems can result in applications with well-fitted supporting software infrastructure, they are burdensome to use in practice. The TinyOS operating system for WSNs [4] uses a more application-driven approach. Applications are written in nesC [5] by wiring high-level OS components together to specify application functionality. The nesC compiler then assembles low-level components that implement the specified functionality. Application-specific partitioning being an automated process is more powerful, but may yield slightly larger compositions. Implementing such systems is challenging, because the analysis may need to extract information from the application not explicitly provided by the programmer. At the back end, this information would have to be utilized to select and instantiate software infrastructure components. A programming language conducive to such program analysis would have to be used, and fine-grained components need to be made available to produce more compact software infrastructure.

Software scaling

While software synthesis yields a compact reified view, it may not be possible to provide some of the selected services and components due to fundamental architectural limitations. In such cases, it is necessary to at least provide these services at a reduced capacity — in other words, provide *scalable service components*.

To illustrate software scaling, take the example of floating point operations. Many embedded processors provide little or no floating point support in hardware. To work around this, floating point values are encoded as large integers, and floating point operations are emulated with integer arithmetic. This effectively provides a floating point unit at the expense of software execution overhead. Conceptually, such a unit would be scaled up to a hardware floating point unit in a more powerful processor. The challenge in scaling a software infrastructure lies in: (i) identifying and representing the design parameters (such as memory, energy efficiency, performance, etc.), (ii) associating the design parameters with application and software infrastructure components, and (iii) using a design tradeoff algorithm in selecting the set of components that satisfy implicit application requirements, and meet overall resource constraints. For example, a time synchronization protocol may be scaled with respect to the temporal precision required and/or residual energy, by using an algorithm modulated by the periodicity of interaction between sensor nodes.

B. Runtime distribution of reified software infrastructure

Reification needs to go a step further and distribute services across the network at runtime. Some of these services (e.g., garbage collection) may be directly requested by the application. Other services such as Just-In-Time (JIT) compilation of hot spots, may be indirectly requested. Most nodes do not have sufficient resources for JIT compilation and storing large symbol tables. Such functionality has to be delegated to more powerful network entities. Service distribution is also necessary due to the inaccessibility of devices. For example, base stations provide services to remotely update, calibrate, test, debug, and instrument the nodes.

This suggests a distributed service architecture in which functionalities and information associated with services are partitioned across base stations and nodes. Otherwise, devices will need to either store the entire suite of software infrastructure, or predict and preload all required runtime services. There are several issues that need to be considered for such a remote service architecture. Services need to be partitioned across nodes. Protocols for subscribing to and delivering services need to be selected. Tightly coupled, or transaction-based interactions provide fine-grained control over device operations, but incur high communication overhead. Scalability is a concern if the base station needs to store node-specific state for a large number of devices. This can be addressed by storing common information on a group basis.

C. Long term application management

In order to support long term management of applications, there must be intrinsic support for managing changes at nodes, and for distributing these updates. This capability can be provided to various degrees depending on the level of sophistication desired. When designing modifiable systems, it is important to abide by carefully considered rules that govern the generation, distribution and application of software changes, in order to guarantee that the system will be in a consistent state at all times.

Update mechanisms can vary in *scope*, *granularity*, *interference*, *security*, and *fault-tolerance*. Scope determines what layers of the software architecture can be affected. Changes in the application should be accompanied by changes in contingent portions of the software infrastructure, so that the software infrastructure can evolve in sync with application software. Granularity refers to the unit of adaptation (e.g., whole system reprogramming vs. statement or procedure-level updates). Interference is quantified by how runtime behavior is affected. Some updates may necessitate a clean reboot of the target, while some may only require a temporary suspension of the running application. Security and fault-tolerance measures are made available through authenticated reprogramming channels, and roll-back mechanisms in the event of a failed update. WSNs require a large scope, fine-grained, and low-interference update model so as to efficiently and flexibly support modifications of both applications and software infrastructure.

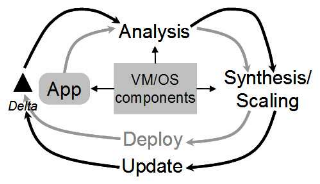

Fig. 2. VM★ deployment cycle.

Figure 2 summarizes the main steps that facilitate the seamless continuity of the develop-update-redeploy cycle during the application's lifetime.

III. VM★

The SENSES software infrastructure is based on the above principles of a common abstraction, software components and reification, distributed architecture, and support for evolution. In this section, we limit our discussion to aspects that are relevant to the guidelines presented in Section II. Implementation details and performance evaluation can be found in [6], [7].

A. Common abstraction

We have chosen the notion of a virtual machine (VM) as the common abstraction for WSN applications. Our VM (VM★) [6] is based on the Java Virtual Machine (JVM) [8] instruction set and Java's programming abstractions. However, the techniques for mapping and reifying the VM abstraction are JVM independent, and can be used for building any VM.

We chose the VM approach for several reasons: First, the VM masks differences among sensor nodes through a common execution framework. Applications can access device-specific features (e.g., sensors or actuators) through a lightweight native interface. Application distribution, management, and interoperability take place through the common intermediate representation. The platform-independent computation model allows code reuse across architectures or generations of architectures. It also enables in-network caching of application binaries, and the use of mobile agents. Second, VM implementations can be customized to exploit specific capabilities of devices. Their implementations can be scaled up or down, can be optimized for specific architectures, or made more resource-aware without affecting applications. Finally, by carefully designing the instruction set, application footprints can be reduced.

B. *VM Reification*

Although VMs are promising candidates for software infrastructure for WSNs, VMs are traditionally either too big to fit on sensor nodes, or have acute limitations in the range of runtime services they provide. For example, SUN's KVM is statically configured and has memory requirements of the order of a few hundred kilobytes. Maté [9] is more acceptable with regard to memory footprint, but provides a limited programming interface and cannot easily accommodate changes to itself after deployment. VM*uses a fine-grained build tool [10] that applies software synthesis techniques to generate instances of VM*that satisfy requirements of specific applications, and applies software scaling on components to respect the constraints of the devices.

VM synthesis involves statically analyzing the application bytecode to determine (i) types, (ii) VM instructions, (iii) runtime services, (iv) device libraries, (v) utility libraries, and (vi) resources required. The synthesis scheme is driven by the insight that the reified VM need only contain the functionality required by the application. Most Java applications only need a small ”working set” of the JVM instruction set. Software synthesis exploits this knowledge to build a self-contained VM that includes implementations of only those opcodes that are used by the applications it will execute. For example, if the application does not use array types, VM array manipulation instructions are not included.

VM scaling involves instantiation of selected VM components and reconditioning them to meet architectural constraints. Scaling algorithms use tradeoffs between various parameters (such as memory, time, energy, responsiveness, coupling between base and sensor nodes) to drive the instantiation of components. For instance, depending on the size of the target device, a suitable object memory layout, memory allocator, garbage collector, etc., are chosen. Devices that have small amounts of memory have a space-efficient object layout, and a garbage collector that sweeps the heap space in a linear fashion to conserve memory.

Synthesis and scaling enable the generation of compact standalone VMs on a per-application basis, for a wide range of devices — from resource-constrained platforms, to intermediate and large systems. By giving the programmer the impression of a complete VM on the sensor node, reification provides expressiveness in spite of the resource constraints of the target.

C. *Distributed VM architecture*

Resource constraints require distribution of VM functionalities, and VM and application meta-information across base stations and more powerful intermediate nodes. VM functionalities that are performed statically and that do not require much interaction with sensor nodes can be pushed to a more powerful node. These include the application configuration, bytecode manipulation, and application and VM static memory management. Static memory management involves managing VM and application code memory, and can be done remotely. In addition, most of the application meta-information (class types, methods, stack configuration, variable types, etc.) and debugging information (line numbers, local variable declarations) can also be stored at the base station. JIT compilation is another example of a service that we implemented as a remote service [11].

D. *Support for long term evolution*

Due to complex behavioral dynamics, application requirements may become clear only after the application has been deployed. Thus, the VM itself may need to evolve by dynamically modifying existing runtime services, or adding new ones. Software updates in embedded processors have typically involved whole system reprogramming (WSP, or reflashing). WSP does not scale to large networks because distributing a binary image throughout the network consumes significant amounts of bandwidth, energy and latency. Also, reflashing at the recipient is a slow and power-hungry process.

VM*'s design fully integrates the notion of change as a key component. The VM is structured so that its components can be selectively modified on the fly in a very fine-grained manner. It allows (i) changing its execution model (e.g., adding or modifying VM instructions); (ii) replacing or adding new runtime services

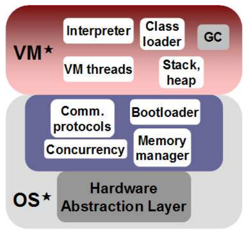

Fig. 3. VM★ architecture.

(e.g., a flash memory manager or a modified routing protocol); (iii) upgrading core VM components for fixing bugs; and (iv) modifying applications through a dynamic class loader. The extensible VM provides an abstract and strict software evolution model that is independent of applications and devices. We provide a more fine-grained update mechanism than WSP, by providing a remote linking service to dynamically link in new or modified components in an incremental fashion. The service is placed on a remote base station because WSN nodes usually do not have resources necessary for linking. The technique also exploits the fact that typical changes to sensor network applications are likely to be small. We have found this approach to be extremely effective in reducing over-the-air reprogramming costs [7].

E. Architecture

Figure 3 shows a simplified overview of the SENSES system software stack, consisting of a low level operating system (OS★) and VM★. OS★ is a lightweight and component-based operating system designed to run on several WSN platforms. It is divided into two core software layers: a Hardware Abstraction Layer (HAL), and a microkernel. The HAL provides an abstraction of the sensor device by implementing device drivers to various resources on the node. For example, it provides interfaces to access the secondary storage, and uses the MCU's hardware timers to supply several software timers to the higher layers. The microkernel implements the ability to bootstrap the VM from the base station or other gateways. It implements a remote bootstrapping protocol that allows the sensor node to download different components of the VM dynamically and securely. The HAL and the microkernel together comprise the operating system (OS★) over which VM★ is built.

VM★ implements a basic VM capable of executing Java bytecodes. It contains an interpreter with its instruction table, core instruction implementations, an object model, system areas and a class loader. Additional components such as the garbage collector can be downloaded if the application requires them.

IV. Conclusion

Although many aspects of WSNs remain topics of debate, we believe that our approach strikes a balance between a number of competing concerns. Most of the experimental deployments of WSNs have been short-lived demonstrations, and this is largely due to the unavailability of suitable software infrastructure. Research efforts to improve the existing approaches would greatly benefit from software frameworks that adopt the guidelines presented in this paper.

It is certain that sensor nodes are going to evolve rapidly, as newer and better subsystems are developed. The individual facets of our approach — VM abstractions, component-based software synthesis, scaling, and incremental evolution — are well known software engineering practices. The key point we make in this paper, is that their cooperated use is critical to the long term viability of system software in WSN deployments.

References

[1] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong, "A Macroscope in the Redwoods," in *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*. San Diego, California: ACM Press, 2005, pp. 51–63.

[2] R. Fjeldstad and W. Hamlen, "Application Program Maintenance Study: Report to Our Correspondents," Tutorial on Software Maintenance. IEEE Computer Press Society, 1983.

[3] B. Lientz and E. Swanson, "Problems in application software maintenance," *Communications of the ACM*, vol. 24, no. 11, pp. 763–769, 1981.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System Architecture Directions for Networked Sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A Holistic Approach to Networked Embedded Systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM Press, 2003, pp. 1–11.

[6] J. Koshy and R. Pandey, "VM★: Synthesizing Scalable Runtime Environments for Sensor Networks," in *Proceedings of the Third International Conference on Embedded Networked Sensor Systems (SenSys)*. San Diego, CA, USA: ACM Press, Nov 2005, pp. 243–254.

[7] ——, "Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks," in *Proceedings of the Second European Workshop on Sensor Networks (EWSN)*. Istanbul, Turkey: IEEE Press, Jan 2005, pp. 354–365.

[8] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, Sept. 1996. [Online]. Available: http://java.sun.com/docs/books/vmspec/

[9] P. Levis, D. Gay, and D. Culler, "Active Sensor Networks," in *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2005.

[10] J. Wu and R. Pandey, "BOTS: A Constraint-Based Component System For Synthesizing Scalable Software Systems," University of California, Davis, Tech. Rep. TR-CSE-2005-18, Aug. 2005.

[11] I. Wirjawan, J. Koshy, R. Pandey, and Y. Ramin, "Balancing Computation and Code Distribution Costs: The Case for Hybrid Execution in Sensor Networks," University of California, Davis, Tech. Rep. TR-CSE-2006-35, Mar 2006.