

IBM Mote Runner—Executive Summary

The core requirements to reap the promised benefits of a fully business-process-integrated infrastructure for deploying large numbers of sensors and actuators are security and end-to-end cost optimizations for such systems. This requires a well-designed ecosystem comprising of inexpensive devices over simple and bullet-proof device programmability to easy integration and use by specialists of the application domain, not of the device technology.

The IBM Mote Runner system addresses these challenges with a high-performance, low-footprint, standards-based software middleware platform comprising a hardware-agnostic and language-independent virtual machine together with development and integration tooling to easily create and manage applications for open sensor and actuator networks.

IBM Mote Runner—White Paper

T Kramp, M Baentsch, T Eirich, M Oestreicher, I Romanov, A Caracas
IBM Zurich Research Laboratory

Status: Thursday, October 16, 2008 — Draft.
Please note that this document is a work in progress that will be updated irregularly.

Abstract

One-way, dedicated data-gathering IT networks, such as those underlying for example UPS' delivery tracking system have shown the commercial value of real-time control of real-world components. Building on this, more generalized applications for wireless sensor networks (WSN) are becoming more and more apparent and significant in size and real-world relevance. Conceptually, the broadest application categories for WSNs provide access and immediate reactions to environmental information, and a flexible communication and intelligence-gathering infrastructure to serve, for example, next-generation business applications that directly tap into the ever greater number of digitally-enabled sensors and actuators providing input to their operation.

Yet, to unlock this potential, two first-order problem categories must be addressed. The first one is the problem of cost: WSNs consist of many small computing elements that have to be cost optimized. In this realm, cost takes the form of up-front investments into the hardware and software plus any subsequent investments (e.g., for maintenance). The former directly translates to hardware cost and with that calls for very efficient software running on the least expensive and cost-effective off-the-shelf chips. The latter translates into design requirements for minimum hardware interaction after mote deployment (e.g., for manual battery change or system reconfiguration). The second problem category is technological: A WSN run-time environment must not only be able to cope with the broad range of technical challenges imposed on WSNs but it must equally be accessible beyond the low-level functionality of individual WSN nodes. Hereby, "accessible" means basically three things. Firstly, it must be possible to dynamically configure and reconfigure the WSN in the field to deal with situations such as interrupted communication or WSN node failures. Secondly, the WSN has to be secured to be considered a trusted source of information and reliable performer of actions in response. Thirdly, the WSN must be well-integrated within the larger infrastructure it cooperates with. It must be generally programmable by domain-specialists to solve domain-specific problems without deep knowledge of WSN technology and components. Only then, real-world aware systems become possible which link—while easy to program and deploy—the physical world of sensors and actuators with business processes and applications, as such improving the responsiveness of business transactions, enabling end-to-end process security, and reducing cost by reaching out into WSNs for data collection, pre-processing, and autonomic feedback.

The IBM Mote Runner run-time environment for wireless sensor networks, currently under development at the IBM Zurich Research Laboratory, tackles these challenges in a holistic manner. Thus, at its core, Mote Runner provides a high-level-language-friendly, resource-efficient and high-performance virtual machine shielding portable applications from hardware specifics. It allows programmers to use object-oriented programming languages and development environments such as C# and Java to develop portable WSN applications that may be dynamically distributed, loaded, updated, and deleted even after the WSN hardware has been deployed. All operations and communications can be cryptographically protected to establish a trusted execution environment. Furthermore, Mote Runner WSN applications provide seamless integration with state-of-the-art back-end infrastructures by means of an event-driven process engine effectively bridging the gap to large-scale business and scientific applications without requiring deep technology skills. Finally, IBM Mote Runner is designed to run on very small, standard, embedded controllers including low-power 8-bit processors thereby reducing both initial investment cost as well as post-deployment and maintenance costs.

Keywords: wireless sensor networks, motes, sensors and actuators, ad-hoc networks, embedded systems, virtual machines, security, ubiquitous business process integration, cost-sensitivity

1. Introduction

A sensor network consists of a potentially large number of spatially distributed autonomous devices, so-called motes [11] that essentially feature sensors and actuators to cooperatively monitor and react to physical or environmental conditions such as temperature, sound, pressure, or motion to name a few. Aside from sensors and actuators, each mote is further typically equipped with a micro-controller, some transient and some persistent memory, a wired or wireless communication device, and an energy source. The envisaged size of a single mote hereby can vary from the size of a small coin to a shoebox while its cost may range from a few dollars to a couple of hundreds of dollars. If the motes predominantly operate wirelessly the sensor network is called a wireless sensor network (WSN). Such a WSN uses radio transceivers and usually constitutes a wireless ad-hoc network, meaning that each mote supports multi-hop routing by forwarding data packets from and to non-adjacent motes. Both star network topologies with a central coordinator as well as more general peer-to-peer mesh and cluster tree networks are used. At the time of writing, the most commonly used network protocol is ZigBee [3] on top of IEEE 802.15.4 [2], providing a maximum over-the-air data rate of 250 kb/s. IP6 over Lower-Power WPAN [4] under specification by IETF and equally relying on 802.15.4 is increasingly attracting interest as an alternative to ZigBee.

The potential of WSNs spans a range of different application domains, exemplified, for instance, by green data centers, intelligent trade lanes, and medical monitoring:

Green Data Centers: Data centers have doubled their energy use in the past five years [1] and thus have reached a tipping point, driven by energy use and cost. From both a cost and an environmental perspective they must significantly increase their energy efficiency in the near future. As a prerequisite, though, it is mandatory to understand a data center's energy use to precisely pinpoint opportunities for its improvement. A WSN deployed throughout the data center could provide detailed temperature readings that can be evaluated using high-resolution 3D temperature maps to enable ongoing energy efficiency assessments. In response, the operation of the data center can be optimized to eliminate hot spots, reduce energy consumption, and overall extend its life time. This optimization even could be partially controlled by the WSN by pre-processing the data collected and making localized autonomic feedback decisions. Computer and building temperature sensors, ventilators, and air conditioning or external air intake ventilators, for example, can be linked to maximize the overall system throughput / energy consumption quotient.

Container Tracking: While new regulations for increases in security in international trade affect all parties of the global supply chain, carriers additionally are looking for new ways to track their goods more precisely and to facilitate data transfer at handover points in the logistics chain. WSN motes connected to containers can establish ad-hoc mesh networks whenever containers are clustered at sea ports or when shipments are discharged to trains or trucks for the transport to their final destinations. In combination with more powerful, stationary motes deployed at clustering points and connected to the Internet, containers can be tracked promptly in this manner. Furthermore, the state of individual containers (e.g., whether opened, tilted exceedingly, dropped from certain heights, exposed to significant temperature changes) can be monitored by means of additional sensors connected to the corresponding container mote and transferred to the carrier at the next clustering point. Provided matching—and entirely new—business models brought about by this technological capability have been established, it might be possible that suitably equipped containers might even negotiate among themselves service guarantees without external intervention.

Medical Monitoring: The field of medical monitoring by itself presents multiple applications for WSNs, from sports medicine to collect data for training and nutrition plans, to the monitoring of elderly people at home or even patients at intensive care units to react more quickly to emergencies. The excessive wiring at intensive care units, for example, makes it difficult to transport patients for special examinations or treatments as the patient's wiring and de-wiring take significant amounts of time during which the monitoring is interrupted or at least severely reduced. At the same time, such an environment imposes extraordinary requirements on the WSN's guaranteed reliability and operation within stringent timing constraints as well as the security and integrity of data communications. As such, medical monitoring represents certainly one of the most-demanding extremes in the spectrum of applications domains for WSNs.

The remainder of this paper is structured as follows: In Section 2, the key challenges lying ahead for WSNs on their way from today's promises to tomorrow's reality are briefly summarized. Next, Sections 3 and 4 present the overall architecture and tool chain of IBM Mote Runner, i.e., its open run-time environment specification and ongoing implementation. Section 5 finally sketches the current status of Mote Runner and outlines next steps.

2. Challenges in Wireless Sensor Networks

Given that WSNs deploy large numbers of motes to gather a view on their environment, costs for mote devices will be a key factor defining initial commercial viability. Cost constraints immediately imply constraints on hardware specifications, namely memory size, computational speed, and communication bandwidth. Thus, low-end, 8-bit micro controllers with 4-8 KB of temporary RAM, 64-256 KB of persistent Flash memory, and data rates of 20-250 kbps will most likely remain predominant configurations in the near future. Such scarce resources require a system design which is foremost guided by efficiency along all dimensions.

Maintenance and maximum lifetime are additional cost factors. Motes tend to be deployed in an ad-hoc fashion and may even change locations autonomously; their exact positions are often not predefined and they may not be easily accessible. Motes are thus required to operate without physical maintenance such as battery replacement for as long as possible, potentially making use of power scavenging to extend their life time [12]. The sheer number of motes deployed may rule out periodic manual maintenance in short term intervals. Scenarios that impose strict constraints on energy consumption, give rise to a whole set of new research and engineering challenges.

Wireless communication in particular requires enormous amounts of energy and therefore demands new protocols especially designed for WSNs. Such protocols foremost must balance the need for short listening periods and a minimum of message exchanges against bounded message latencies and bandwidth guarantees. Due to uncertain radio connectivity, battery drain, device lockups, or physical tampering, motes within WSNs are further considered to be generally unreliable. Even during normal operation, motes may sleep for long periods of time to conserve energy, not being able to communicate during these periods. Protocols for WSNs therefore must be able to cope with (temporarily) unavailable motes and transient periods of network separation by reconfiguring dynamically.

While the aforementioned aspects are generally considered as the economic make-or-breaks of a sound and solid foundation of WSNs, it is the tackling of technology challenges that will eventually decide if, when, and at what rate WSNs will eventually become widely adopted.

Firstly, as the number of applications for WSNs grows and the range of application domains broadens, flexible configuration and dynamic reconfiguration in the field becomes increasingly important. This includes the ability to update applications deployed even without physical access; to re-use or augment a WSN infrastructure for a different / additional application, and to share existing WSN infrastructures among multiple applications. This requires abstracting from the real, usually rather heterogeneous and often still proprietary, non-standardized selection of mote hardware components—processor, memory, radio network as well as sensors and actuators—and providing a unified, virtualized environment for application execution. Furthermore, a layer that provides dynamic code distribution, loading, updating, and deletion is needed which allows multiple applications to co-exist.

Secondly, as more and more applications rely on the data provided by a WSN's sensors and on the execution of commands directed at its actuators, WSNs must be established as trustworthy sources of information and reliable performers of actions. For this, the motes themselves and all communication must be protected against both active and passive attacks to ensure that data received from and commands sent to the WSN are not compromised. WSNs further should be robust against, or at least gracefully recover from, denial-of-service attacks which in turn may partially build upon mechanisms dealing with partial and temporal mote failures. In "federated" environments where multiple parties want to share a single WSN, motes must also be protected against applications trying to monopolize the WSN, and some distributed rights management must be put in place to establish the rules according to which applications may be distributed across, loaded on, and deleted from the WSN and by whom.

Thirdly, as WSNs usually do not stand alone but are the "eyes and ears" (and sometimes "fingers") of much larger applications, they must be well-integrated with the application's infrastructure. This could be done, for instance, by means of a standards-compliant business process engine that interfaces the WSN with the backbones of today's large-scale enterprise and scientific applications. At the same time, WSNs must become generally programmable by the domain-specialists who are developing those applications. WSNs thus have to evolve into tools whose full potential can be tapped even by non-WSN experts. Significant efforts are required to achieve such a level of usability, not only regarding the development tools and their integration into application-domain specific environments but even more by establishing algorithms and abstractions that allow WSNs to be programmed as single units. For example, when reading a temperature value, it is often only relevant that the value is from within some specific area, not that it has been

read by one specific mote. The automatic mapping of such higher-level application-specific criteria to motes becomes even more crucial if the motes within a WSN may change location over time.

Fourthly, in consideration of what has been mentioned before, it must be easy to trade off cost factors against each other to arrive at the most economical overall solution. If, for example, a lack of sufficient data pre-processing capabilities within a WSN leads to higher data loads on subsequent processing locations, the savings obtained by deploying less expensive motes may be quickly offset by more expensive networking and back-end processing requirements. Another example in this realm deals with the level of expertise required to deploy a certain type of application: One may be tempted to deploy a network of sensors and actuators using proprietary processing and communications paths judging only the purchasing costs of those components. If maintenance and evolution costs, exemplified by programming new functionalities, are taken into account, a more flexible and standards-based infrastructure is likely to be more economical.

3. IBM Mote Runner

IBM Mote Runner is both a run-time environment specification and implementation for small embedded systems, primarily WSNs, that aims at tackling all the aforementioned problems. On the one hand, this happens directly, by providing a unified, virtualized execution environment for flexible configurations. On the other hand, this is done indirectly, by providing the infrastructure and mechanisms for the implementation of higher-level and application-domain-specific solutions. From an engineering perspective, the Mote Runner specifications are designed to be implemented in portable, scalable, and efficient ways, building upon and carrying forward our experience with JCOP, the IBM JavaCard implementation [5].

The Mote Runner implementation builds upon off-the-shelf embedded (mote) hardware with a thin hardware abstraction layer written in C and Assembler encapsulating any hardware-specific functionality as illustrated in Figure 1. At the next layer, there is a virtual machine (again written in C), run-time library (written in C and C#) and 802.15.4 MAC layer (written in C). Both the run-time library and the MAC layer expose APIs for application development in higher-level languages such as Java and C#. Higher-level network communication protocol stacks are implemented on top of the run-time library and the MAC layer, in turn exposing further APIs to the application programmers.

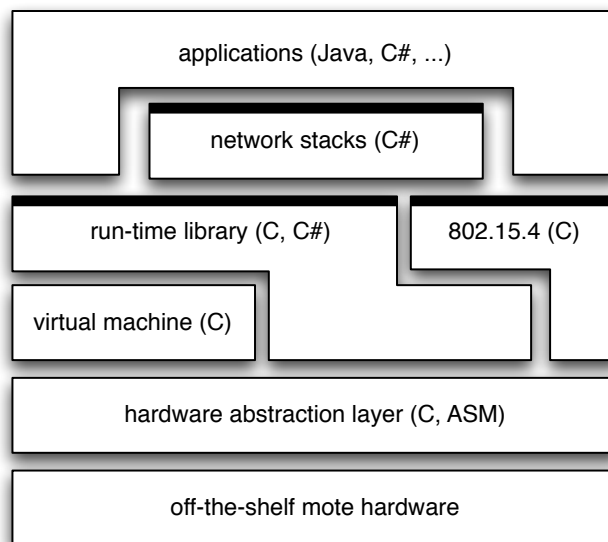


Fig. 1: The Mote Runner layered architecture

Portability

In Mote Runner, portability generally works at several levels. Firstly, by introducing a virtual machine, applications are shielded from the hardware particularities of a broad range of different motes. Specific features of individual motes can still be made available via optional interfaces while their presence or absence can be listed in a profile describing a mote's configuration at a given point in time. Application development then becomes independent from individual mote hardware configurations, being more tied to a mote's feature set and potentially its non-functional characteristics instead.

Secondly, as the specifications of the virtual machine byte code, the application load format, and the application programmer's interface (API) will be made openly available, applications developed against these specifications can be executed on different implementations of the Mote Runner run-time platform specifications. Application development then becomes independent from a specific Mote Runner implementation which allows developers not only to choose the implementation which provides the best match but also to gracefully switch to a better matching Mote Runner implementation if either such an implementation becomes available or the application's requirements change over time.

In our implementation of Mote Runner, we further focus on a third level of portability, namely the ability to implement the run-time platform in a way such that it can be made available on some new mote hardware with as little effort as possible yet without sacrificing efficiency. This is achieved by a combination of compile-time switches to configure characteristics such as the platform's endianness or memory layout, and design decisions such as the introduction of a small hardware-abstraction layer. Furthermore, most higher-level code is developed in C# and translated to Mote Runner byte code for compactness and easier maintenance.

Scalability & Efficiency

Even though WSN motes vary broadly in terms of processing power and the amount of memory available, small devices with scarce resources are predominant. Powerful motes usually serve only as hubs or gateways, interconnecting clusters of less powerful ones or integrating them into a larger infrastructure. Thus, run-time environments for WSNs must be designed from the ground up to run efficiently on motes with very scarce resources, that is, 8-bit micro-controllers with only a few kilobytes of transient memory and tens of kilobytes of persistent memory, while still making good use of additional resources if available. In general, it tends to be significantly easier to scale up and make use of more resources than the other way round.

For Mote Runner, we therefore did not simply decide to use a subset of an existing byte code such as one of the many Java variants or Microsoft's Common Language Runtime. These byte codes are either too demanding in terms of processing power and memory (e.g., promoting all values internally to 32 bit values), or have been found significantly lacking in functionality based on our experience with the development of a Java virtual machine for smart cards, or are too much tied to one particular language and/or development model. Furthermore, the often-heard claims of being able to execute embedded applications on standard PCs are basically void as application areas differ significantly resulting, for example, in fundamentally different API semantics. Instead, Mote Runner uses its own simulator executing literally the same virtual machine as any mote with a PC-specific hardware abstraction layer. As such, a very high level of compatibility between "real motes" and the simulation can be achieved allowing for convenient mote application development and network simulation.

Virtual machine and system services are further adopted to match the scarce resources and unique characteristics of motes. These include the memory model, garbage collection, driver interfaces and remote management. For instance, dynamically loading and linking an application in standard virtual machine environments requires a significant amount of temporary information and state. In contrast, Mote Runner specifies a load format that allows a mote to stream link an application while loading without temporarily buffering any data.

Higher-Level Languages

From a higher-level language point of view, the only requirement on the Mote Runner byte code is the ability to map it from a higher-language with a reasonable default behavior and without sacrificing efficiency. That is, within these limits Mote Runner is language-agnostic. This is achieved by specifying a typed byte code which is both compact and can be executed efficiently on very small platforms. While this restricts support for higher-level languages to those that provide type information at compile time, avoiding run-time lookups of type information is mandatory to execute efficiently on cost-effective 8-bit micro-controllers. In addition, many languages such as Javascript that generally support dynamic type information nowadays allow optional type information to be provided by the programmer.

Since the processing of 32-bit data types puts significant burden on 8-bit micro-controllers, higher-level language integer data types will be mapped to 16 bit integers. While 32-bit integers can still be explicitly used by means of a higher-level long integer data type, the default of a 16-bit data type tends to avoid making the platform unnecessarily slow by programmers not savvy writing efficient code. The default 16-bit data type reflects the native capabilities of the targeted processors which should not be hidden from the application programmer. Mote Runner initially specifies language subsets for Java and C# (e.g., no thread support) including accompanying mappings of C# and Java data types

onto Mote Runner byte-code data types [7]. Correspondingly, our implementation provides support for C# and Java mote application development including integration into Microsoft Visual Studio and Eclipse, respectively.

Thirdly, for interoperability, all system and library APIs are specified and distributed only in a language-independent format from which stubs for different higher-level languages such as C# and Java can be generated on demand. The expressiveness of the interface specification format is sufficiently rich to retain the semantics of the higher-level language subsets and type mappings specified by Mote Runner.

Event-Driven Programming Model

Embedded systems in general and WSNs in particular operate inherently reactive, driven by events such as changes in the environment captured by sensors, time interrupts, or incoming messages from neighboring motes. In response, some processing is triggered, further events may be generated, and certain actuators may be activated. Mote Runner therefore builds upon an event-driven programming model instead of providing threads, which firstly leads to a more natural way of structuring applications and, secondly, is more efficient and resource-friendly. Naturally, as mentioned before, threads are then also left out from the Mote Runner language subsets defined for Java and C#. At the level of C#, Mote Runner APIs make extensive use of delegates to represent and invoke event handlers. In Java, delegates are emulated by some specific coding patterns which can be identified during compile time and mapped onto the corresponding delegate byte codes of the Mote Runner virtual machine.

Remote Management

One of the primary goals of Mote Runner is to provide an infrastructure that allows applications to be dynamically distributed, loaded, and updated as well as deleted from a live WSN. The virtual machine is one key component for this, as it provides a foundation which facilitates executing the same binary code on all motes of a heterogeneous WSN. In conjunction with a load-file format suitable for stream linking, new applications can also be stream distributed across the WSN to save resources. To distribute an application throughout the entire WSN, it can be injected via virtually any mote and from there flooded to the other motes. Each packet received by a mote can be forwarded to neighboring motes immediately without the need for the whole application being temporarily stored in distribution format.

Of course, the ability to dynamically change the set of applications available on a WSN automatically raises the questions of who is allowed to distribute, load, update, and delete an application, and how to ensure that only “approved” applications are installed to protect the WSN’s integrity. The former question can be tackled by introducing a delegated management scheme based on public-key cryptography as already successfully demonstrated by Global Platform for JavaCards. To address the latter question, either applications could be digitally signed off-mote and subsequently motes only accept applications with a valid signature for distribution and loading, or an on-mote code-verification scheme could be used as, for instance, by Java. Code verification, though, again requires quite some resources which are simply not available on smaller motes. Yet a combined approach in which more powerful motes perform on-mote code verification and digitally sign approved code prior to distribution to less powerful motes would be possible and may make sense for certain environments. Mote Runner therefore actively explores both routes.

4. Mote Runner Tool Chain

In addition to the aforementioned implementation of the run-time environment, IBM Mote Runner comprises a whole tool chain which is used both for the development of the run-time environment and for application development likewise. All the steps from the source code to a binary application ready for distribution are illustrated in Figure 2.

After the source code has been written in any one of the higher-level languages supported by Mote Runner (i.e., currently C# and Java), all the source code is compiled by any compiler available for the specific language used. For Java this is typically javac whereas for C# this may be either the Microsoft or the Mono C# compiler. The output generated by these compilers, namely class files for Java and dll files for C#, is then read by the converter and translated into assembler source code, the so-called Mote Runner intermediate language (SIL). Next, the assembler processes these files plus all the exchange files (.sxp) of any external libraries (including the system) the new assembly depends on, and generates either a debug (.sda) or binary (.sba) assembly for debugging or distribution, respectively. Furthermore, an exchange file describing the public interfaces of the just compiled assembly is generated to be referenced by other applications in order to permit others to build on previously created code. The stub generator in turn outputs higher-level language stub files for Java and C# from these exchange files as required, closing the circle from the higher-level language to the intermediate language and back thus obviating the need for language-specific header files.

While these tools are self-contained, Mote Runner also provides debugging plug-ins for Microsoft Visual Studio and Eclipse to allow C# and Java source-level debugging of mote applications in conjunction with the Mote Runner simulator. The simulator executes literally the same virtual machine and run-time environment as any physical mote, just relying on an extended hardware abstraction layer for integration with an external debugger. The simulator further provides various interfaces for remote mote management to load and delete applications dynamically.

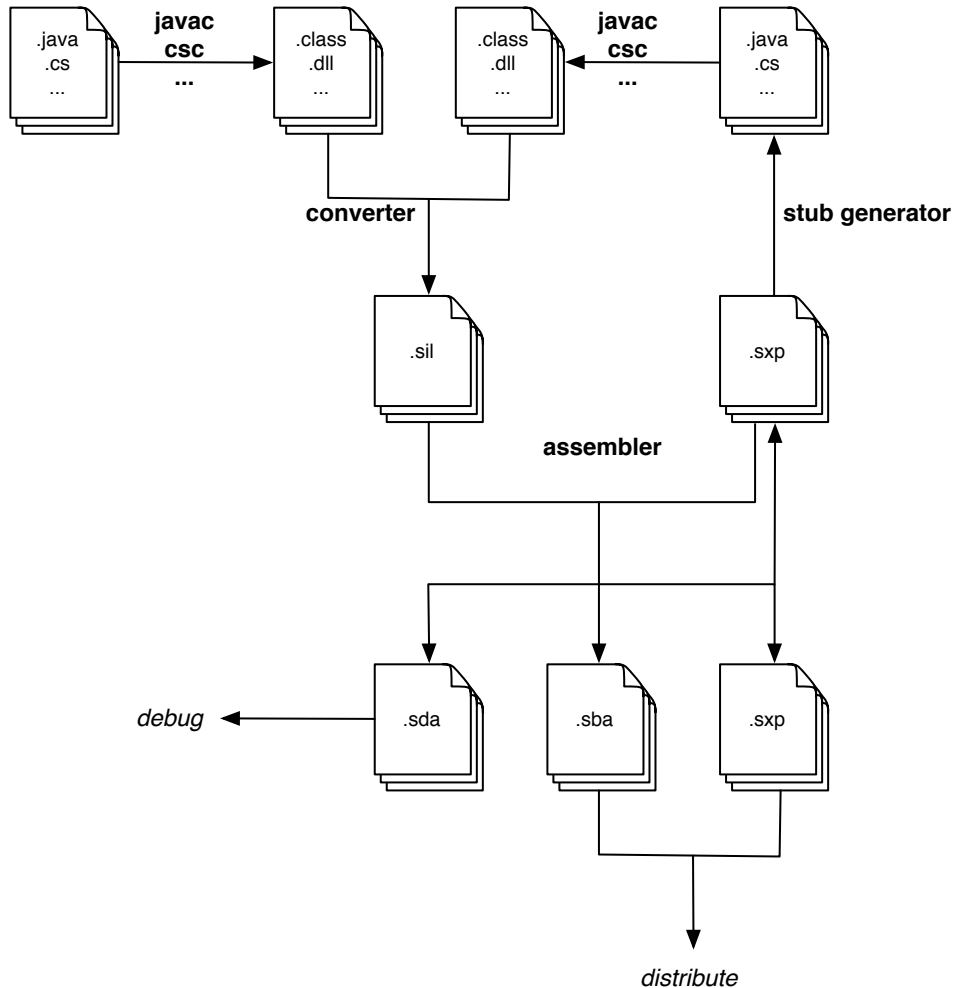


Fig. 2: The Mote Runner tool chain from source code to binary applications.

5. Related Work

Related work in the field of run-time environments for wireless sensor networks can be classified mainly into two categories: native platforms and virtual machines.

Native Platforms

The most widely used native platform for sensor networks is TinyOS [6], an open-source operating environment initially developed by the U.C. Berkeley EECS Department and backed by the TinyOS Alliance nowadays. It comes with its own C programming language dialect called nesC that extends C by introducing an event-driven execution and concurrency model paired with component-oriented application design. TinyOS has been designed with executing a single application per mote in mind. Applications are linked with an operating system image and, as needed, additional libraries to create an executable image to be loaded onto a specific mote.

While such a setup tends to make good use of the available resources, the omission of run-time checks (e.g., array boundary checks, type safety) and leaving low-level tasks such as memory management to the programmer is error prone. In addition, implementing an application across heterogeneous mote hardware requires not only recompiling for each platform but potentially additional work, not to mention the burden of maintaining different code bases afterwards.

To update an application it is furthermore necessary to build a completely new image and reprogram each mote individually. As this is sort of a limiting factor in more dynamic environments, the Maté [8] virtual machine running on top of TinyOS has been developed. Maté offers a small set of assembler-like instructions specifically tailored for the implementation of sensor networks applications which then can spread virally in a WSN. Programming in Maté is rather low-level which significantly limits its appeal to a broader audience.

Virtual Machines

As an alternative to native platforms, virtual machines are increasingly touted to cover the hardware heterogeneity of and deal with the flexibility requirements imposed by next-generation WSNs. The Java programming language on top of the CLDC specification [10], in particular, is gaining momentum right now, with Sun's Squawk virtual machine running on SunSPOT motes [6] and the recent Sentilla VM for the JCreate motes [9] serving as prime examples. Basically, the CLDC eliminates user-defined class loaders, thread groups and daemon threads, finalization of class instances, errors and asynchronous exceptions from the Java VM specification and defines a minimum set of system and utility classes. Application management is left to the profile implementations on top of CLDC.

Even the minimum resource requirements of the CLDC specification are relatively high and not well-suited for tiny sensor motes equipped with an 8-bit processor, less than 128 KB non-volatile memory, and 4-8 KB of volatile memory. The SunSPOT platform, for instance, mostly resembles a mobile phone or PDA in terms of computational power and memory capacity (i.e., using a 32-bit ARM processor with 4 MB of non-volatile memory and 512 KB of volatile memory), while Sentilla's JCreate motes are equipped with a 16-bit micro-controller in turn.

In contrast to the event-driven programming model of TinyOS, the CLDC relies on multiple threads for monitoring sensor values, performing computations, and communicating via radio. Support for multiple threads comes at a price, however, as the stack state of each thread requires volatile memory space. Moreover, context switching between threads imposes significant execution overhead especially on tiny platforms.

References

- [1] Koomey, Jonathan. *Estimating total power consumption by servers in the U.S. and the world*. Analytics Press. February 2007. <http://enterprise.amd.com/us-en/AMD-Business/Technology-Home/Power-Management.aspx>
- [2] IEEE. *802.15 WPAN Task Group 4 (TG4)*. <http://www.ieee802.org/15/pub/TG4.html>
- [3] ZigBee Alliance. <http://www.zigbee.org>
- [4] IETF. *IPv6 Over Low Power WPAN (6lowpan)*. <http://www.ietf.org/html.charters/6lowpan-charter.html>
- [5] M. Baentsch, P. Buhler, T. Eirich, F. Hoering, M. Oestreicher. *JavaCard — From Hype to Reality*. IEEE Concurrency 7, 4, pages 36-43, October 1999.
- [6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. *System Architecture Directions For Networked Sensors*. Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pages 93–104, 2000.
- [7] T. Kramp, M. Baentsch, M. Oestreicher, A. Caracas, T. Eirich, I Romanov. *A Multi-Language Virtual Machine for Embedded Devices*. Submitted to Eurosys 2009.
- [8] P. Levis, D. Culler. *Maté: A Tiny Virtual Machine for Sensor Networks*. Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), pages 85–95, 2002.
- [9] Sentilla. <http://www.sentilla.com>
- [10] Sun Microsystems. *Connected Limited Device Configuration: Specification Version 1.1*. <http://java.sun.com/products/cldc>
- [11] Sensor Node. http://en.wikipedia.org/wiki/Sensor_node.
- [12] D. Steingart, J. Polastre. *Energy Harvesting White Paper*. http://www.sentilla.com/pdf/Sentilla_Energy_Harvesting.pdf