Enabling Java™ for Small Wireless Devices with Squawk and SpotWorld

Randall B. Smith
Sun Microsystems Labs
16 Network Drive
Menlo Park, CA 94025
randall.smith@sun.com

Cristina Cifuentes
Sun Microsystems Labs
16 Network Drive
Menlo Park, CA 94025
cristina.cifuentes@sun.com

Doug Simon
Sun Microsystems Labs
16 Network Drive
Menlo Park, CA 94025
doug.simon@sun.com

ABSTRACT

We report on a project at Sun Microsystems Laboratories investigating small wireless transducer systems. The project includes many facets: we're designing and building the hardware, porting a J2ME compliant Java VM called “Squawk” to run on the devices, creating transducer I/O, networking, and security libraries, building programming tools, and making example applications. Here we focus on the tool environment called SpotWorld and on the Squawk virtual machine. Squawk runs on conventional networks of desktops and servers as well as on wireless networks of small transducers, and so enables a unification of those two domains. We believe combining Java with the proper tools will make working with wireless device networks significantly easier.

1. INTRODUCTION

The pervasive computing vision depicts a future in which computation is widely embedded in the everyday world. One medium for enabling this vision is the tiny, wireless computer that connects to the world with sensors and actuators. At Sun Microsystems Laboratories, we have been investigating wireless transducer networks by creating our own device we are calling the Sun™ Small Programmable Object Technology, or Sun SPOT (See Figure 1). The Sun SPOT main board is based on an ARM7 processor that includes 2MB of flash memory and 256KB of SRAM, plus a separate 802.15.4 radio chip. Additional sensor boards can be attached: the “demo” sensor board includes a 3-axis accelerometer, a light sensor, a temperature sensor, two push buttons and 3 LEDs (two of which are tri-color). The demo sensor board also provides access to 9 general I/O lines.

With the emergence of processor chips accompanied by such large amounts of memory, we believe it is reasonable to use a higher level language such as Java to program such devices. Java brings with it garbage collection, pointer safety, exception handling, and a mature thread library with facilities for thread sleep, yield, and synchronization. Furthermore, Java's object-oriented facilities match well to a domain of physical devices, which can be organized into a hierarchy of general to specific functionality leveraging inheritance and polymorphism.

In addition to Squawk, we are providing tools for programming. deploying, and monitoring these devices. Part of the difficulty in working with today's wireless sensor networks arise from the lack of user interface on the device. We are creating tools that visually represents each Sun SPOT on a desktop screen, so that the user can monitor each application, inspect and edit the code, and (re)deploy applications. Because Java also runs on desktops and servers, the same tools can be used to visualize the portions of distributed applications that reach into the conventional network.

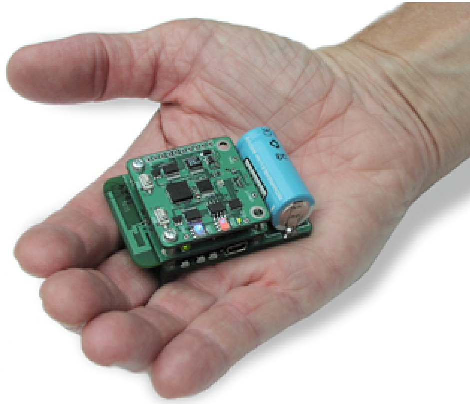

Figure 1: The Sun SPOT is a small wireless transducer. Here we see the demo sensor board on top, with the main processor and radio board in the middle. The bottom board is a battery board that also provides a reset button and USB connector with indicator lights to show battery charge status and USB traffic.

2. THE SQUAWK JVM

Squawk [1] grew out of earlier efforts at Sun Labs on systems such as the KVM [2] to create a CLDC, J2ME compliant virtual machine engineered for small devices. Like some other JVMs [3, 4, 5, 6], Squawk is written almost entirely in Java. Akin to [4, 5], Squawk is designed to run on the bare metal -- no OS footprint is required. In addition, Squawk is specially designed to run on small, memory constrained devices. As Squawk is mostly written in Java, certain space savings arise automatically because bytecodes are a more efficient representation than the equivalent functionality in native code. And being almost all Java, porting to new platforms is made easier.

Most Java VMs run a single application, whereas a Squawk VM can run multiple applications. In order to achieve this, Squawk employs an application isolation mechanism [7], so that per-application mutable state is not shared between applications. Each application or *isolate* in Squawk maintains its own copy of mutable state. Hence, the initialized state of classes and the values of static fields are isolate-local. Also, synchronization on shared immutable resources (e.g. class monitors) is tracked on a per isolate basis so that the threads within one isolate are managed as a group without affecting threads in another.

Squawk achieves further space savings by verifying class files in advance so that a class verifier is not required inside the VM. The preverifed classes get bundled into what are called *suites* for deployment. Furthermore, each suite is also an especially memory efficient representation of the classes it contains. On average, suite files are 35% the size of classfiles.

The compounded memory savings deliver a truly small footprint Java VM: The current version of Squawk running on the Sun(TM) SPOT uses 80KB of RAM and 270KB of flash memory, including all the libraries for CLDC 1.0, radio, and sensor / actuator access. This makes it suitable for recently emerging single chip devices at the high end of the integrated processor+memory spectrum.

In addition to the memory footprint space saving features, Squawk has two other features making it an interesting research platform especially suitable for small devices.

Isolates as objects: In Squawk, each application is represented by a Java object. This object is an instance of class Isolate, and can be used to query the status of the associated application, and even directly affect that application through methods such as start(),pause(), resume(), and exit(). These objects naturally facilitate creating tools that inform the programmer of which isolates are running, about the status of each isolate, and enables the developer to manage the deployment by interrupting the execution, resuming it, or simply quitting the application.

Isolates can also be stored onto disk, to checkpoint the application as it runs. This means that the status of each thread, including all temporary variables, can be serialized onto a stream for storage. This is easier to accomplish in Squawk because the entire state of an application, including thread stacks) is represented as Java objects. In many conventional Java VMs, the stack associated with each thread is not represented as a Java object, so serialization becomes more difficult.

Isolate Migration: Because each application can be serialized onto a stream, one can imagine reading that stream into another Squawk VM to immediately reconstitute the isolate, skipping the step of storing onto disk. This effectively migrates the isolate between VM's. Migration of isolates could enable a new class of application, one that wanders through the network as part of accomplishing its computational task.

We have started simple experiments with isolate migration, and can currently move running applications from one Sun(TM) SPOT to another, or from one desktop or server to another. When moving to an architecture with a different endianness, the appropriate translation is performed, as we migrate a literal object graph which is represented as binary data.

Isolate migration and checkpointing may be reminiscent of similar facilities available in, say, Smalltalk snapshots. In that case the snapshot may wake up on a different machine, with different Ethernet address, different display size, and several other different hardware capabilities. In Squawk however, an individual application is the granularity of serialization. Before moving, the isolate must close all open connections to the external world and record relevant information, so that upon waking it can restore the connections. The waking isolate must sniff out the new environment, and reconnect accordingly. In principle, it may not be possible to successfully connect to the new environment. Thus we expect isolate migration to be utilized by programmers in specific situations where such problems are known to be manageable.

We have added a moveTo(IPAddress ip) method for isolates, to facilitate our early experiments. With this method, an application can itself decide to move from one device to another. We expect this facility could be used for load balancing, or for scripting a single client server application that moves rather than writing two applications that connect. Isolate migration could be especially useful to affect an in-the-field replacement of one device by another (e.g.: with fresh batteries) by letting the user simply pull the software from the old device onto the new. A summer intern wrote an application that migrated itself home upon encountering an exception, so that it could be debugged on the programmer's workstation before being sent back into the field.

3. THE SPOTWORLD ENVIRONMENT

We are also working on desktop tools to facilitate programming, monitoring, debugging, and deployment. The main desktop tool, called SpotWorld (see Figure 2), can run standalone or as pane inside an IDE (currently we are supporting the NetBeans IDE). Desktop Squawk programs can communicate with the local wireless devices through a Sun SPOT that is connected to the desktop with a USB cable, and that is running a special BaseStation program. The desktop applications can then open a “radio” channel just as would other wireless devices, only the channel is transparently routed through the BaseStation and out over the BaseStation's radio.

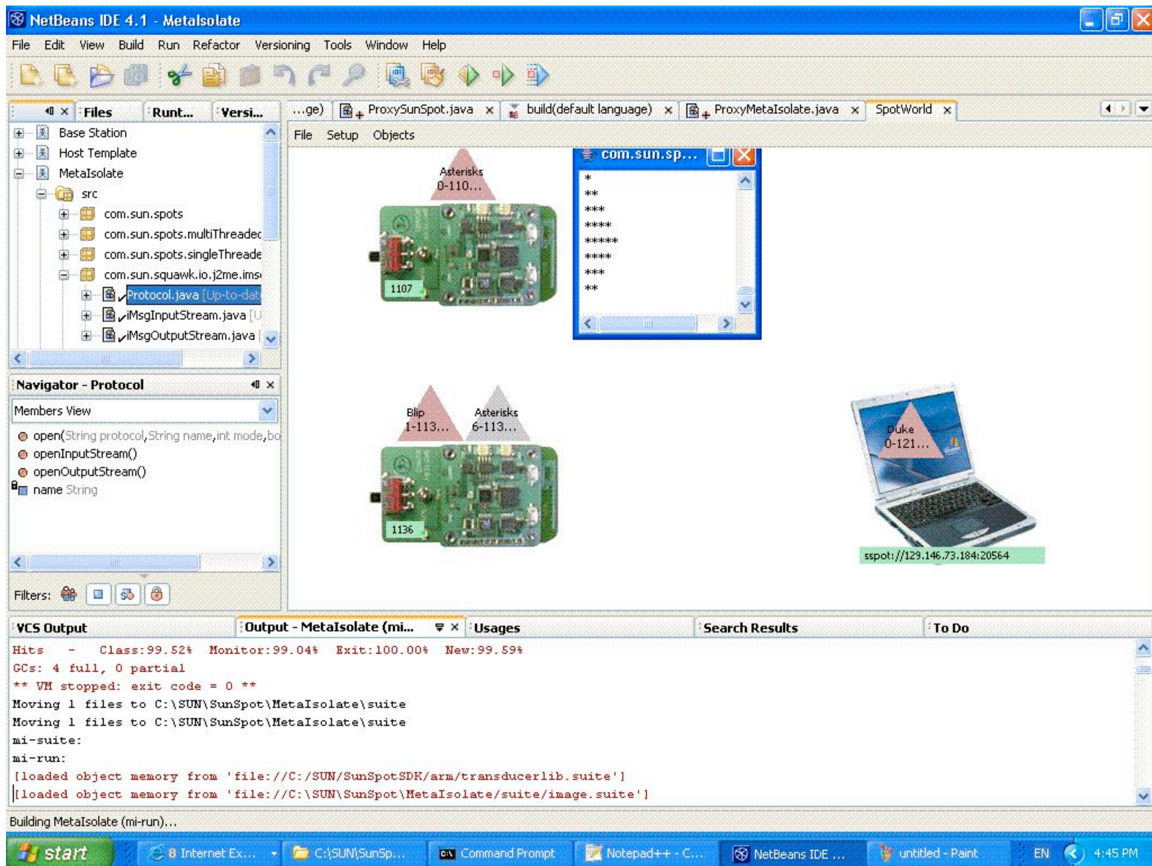

Figure 2 The SpotWorld environment is shown here running inside an IDE. Three devices, each running Squawk are shown here. Triangular icons represent applications running on the device. The small window near the upper Sun SPOT represents the System.out stream from the application running there.

In order for SpotWorld to depict the various instances of the Squawk VM, and to let the user manage applications running on each, SpotWorld must discover the instances of the VM and communicate to them with some agreed upon interface. Consequently, each VM runs an application called the MetaIsolate that is listening for connections to any SpotWorld instances that may wish view or manage applications on the VM.

The MetaIsolate appears on the SpotWorld screen as an icon that looks like the hardware upon which it runs. This may be a server, desktop, laptop, or Sun SPOT. The MetaIsolate responds to a ping-like protocol to facilitate discovery. It also allows new instances of applications to be remotely started. When this happens, SpotWorld causes a triangular icon to appear on the screen, animating into place atop the depicted device.

The user can pop-up a menu on each triangular isolate icon. This menu contains commands such as pause, restart, exit, or edit. Editing causes the embedding IDE to open a code editor pane on the application. The user can edit then redeploy and restart the application from within the user interface.

The user can also view the Java “System.out” text stream for any application within SpotWorld. This is supported by special extensions to Squawk that allows System.out to be multiplexed out across arbitrary streams, including the radio. Seeing application output can be especially useful when debugging isolates.

As mentioned above, Squawk supports migration of running applications. This is supported in SpotWorld by drag and drop. The user can pull a triangle off one VM and deposit it on another, where the application will continue running where it left off. This is of course not always possible, as the target device may be incompatible with the assumptions of the code, or may require the isolate to disconnect and reconnect cleanly, and we have not yet implemented such facilities.

We also hope to support standard Java debugging protocol from within SpotWorld, as well as some sort of read-eval-print type-in shell for inspecting and managing the Sun SPOTs. We would also like to support simulated Sun SPOTs: a user might start development of a distributed application by working solely on the desktop with simulated wireless devices, then incrementally migrate each isolate into place on a real device, testing deployment at each step of the way.

4. CONCLUSIONS

We have presented an overview of the Sun SPOT system, with emphasis on the Squawk Java VM and the SpotWorld tools used to program and manage the devices. This work brings the benefits of high-level, object-oriented languages to the world of small wireless devices, including standard development tools programmers use in their desktop environment. As hardware capabilities continue to increase, the extra marginal cost in size and power necessary to support this approach will diminish. We imagine a future in which a single programming paradigm will apply to both the wired and wireless networks.

5. RERERENCES

[1] D. Simon and C. Cifuentes The Squawk Virtual Machine: Java™ on the Bare Metal. To appear in Companion Proceedings ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), San Diego, California, Oct. 2005. ACM Press. Also see http://research.sun.com/projects/squawk

[2] A. Taivalsaari, B. Bill, and D. Simon. The Spotless system: Implementing a Java(TM) system for the Palm connected organizer. Technical Report SMLI TR-99-73, Sun Microsystems Research Laboratories, Mountain View, California, Feb. 1999.

[3] B. Alpern and D. A. et al. Implementing Jalapeno in Java. In Proceedings ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Denver, Colorado, Nov. 1999. ACM Press.

[4] M. Golm, M. Felser, C. Wawersich, and J. Kleinoeder. the JX operating system. In Proceedings of the USENIX Annual Technical Conference, pages 45–58, Monterey, CA, June 2002.

[5] S. Lohmeier. Jini on the Jnode Java OS. Online article at http://monochromata.de/jnodejni.html, June 2005.

[6] K. Palacz, J. Baker, C. Flack, C. Grothorff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In Proceedings of IVME, 2003.

[7] G. Czajkowski Application Isolation in the Java™ Virtual Machine. In Proceedings ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Minneapolis, Minnesota , Oct. 2000. ACM Press.