Towards a Distributed Java VM in Sensor Networks using Scalable Source Routing

[Extended Abstract]

Bjoern Saballus
University of Karlsruhe
Am Fasanengarten 5
76131 Karlsruhe
saballus@ira.uka.de

Johannes Eickhold
University of Karlsruhe
Am Fasanengarten 5
76131 Karlsruhe
jeick@ira.uka.de

Thomas Fuhrmann
Technical University Munich
Boltzmannstrasse 3
85748 Garching
fuhrmann@net.in.tum.de

ABSTRACT

One of the major drawbacks of small embedded systems such as sensor nodes is the need to program in a low level programming language like C or assembler. The resulting code is often unportable, system specific and demands deep knowledge of the hardware details. This paper motivates the use of Java as an alternative programming language. We focus on the tiny AmbiComp Virtual Machine (ACVM) which we currently develop as the main part of a more general Java based development platform for interconnected sensor nodes. This VM is designed to run on different small embedded devices in a distributed network. It uses the novel scalable source routing (SSR) algorithm to distribute and share data and workload. SSR provides key based routing which enables distributed hash table (DHT) structures as a substrate for the VM to disseminate and access remote code and objects. This approach allows all VMs in the network to collaborate. The result looks like one large, distributed VM which supports a subset of the Java language. The ACVM substitutes functionality of an operating system which is missing on the target platform. As this development is work in progress, we outline the ideas behind this approach to provide first insights into the upcoming problems.

Keywords

distributed Java VM, sensor networks, embedded systems

1. INTRODUCTION

Today's life is hard to imagine without mobile phones, PDA's and other small devices. Additionally, everyday's life devices such as toasters and refrigerators are equipped with an increasing amount of computational power. As most of the usual tasks of these devices hardly exhaust this computational power, there is room to implement new, more sophisticated tasks. This idea becomes even more interesting if all these devices will be equipped with some kind of communication interface which allows to exchange data and trigger actions on remote nodes. The result is a distributed network of communicating small, embedded devices which belongs to the field of ubiquitous computing and ambient intelligence.

The BmBF research project *AmbiComp* aims at interconnecting all the above mentioned everyday life devices in an ad-hoc manner. Although the idea originated in the field of "digital" or "intelligent" homes, it extends to a much broader view of distributed computing: the devices will share data and distribute their workload among each other. The vision will be leveraged by the algorithms and protocols that we will develop in this project.

The paper is structured as follows: Section 2 gives an overview over other projects that have similar goals as the AmbiComp project. In section 3 we describe our novel AmbiComp VM, its design rationale and the tool chain from the Java file to the running program on the ACVM. Section 4 outlines how the ACVM addresses code and object migration. Finally, section 5 concludes with a summary of this paper.

2. RELATED WORK

Distributed Java VMs have been addressed in the literature before. Until now, most of these works originate from the fields of cluster computing and high performance computing. For example, Zhu et al. [9] propose JESSICA2, a distributed Java virtual machine (DJVM) which uses a global object space and transparent Java thread migration to provide a single system illusion. Haumacher et al. [5] describe how Java's remote method invocation (RMI) can be used to create and control transparent distributed threads in their JavaParty system.

In contrast to these heavy-weight approaches, we want to explore how they can be applied to embedded systems, for example, in sensor actor networks. To this end, our starting point is a small footprint Java VM. From its beginning, Java has been associated with the idea of cross-platform programming of embedded devices. Especially, the Java Micro Edition sets the focus on VMs that shall run on mobile devices like cell phones, PDA's etc.

Here, we give an overview of these VMs including their advantages and disadvantages concerning their use on even smaller embedded devices like sensor nodes. The criteria for this comparison are:

1. What are the specifics of the target platform hardware (processor, flash-rom, ram and clock speed) the VM is supposed to run on?
2. To which extent does the VM support the Java language specified by the Java Language Specification [3]?
3. Which set of class libraries is supported by the VM?

Beside the ACVM, the compared VMs are the *KVM* (Java ME: CLDC 1.1), *Squawk VM* (Sun SPOTs), *NanoVM* (Robots: Asuro) and the *ParticleVM* (Particles).

The KVM is a CLDC reference implementation implemented in ANSI C. It is the predecessor of the CLDC HotSpot Implementation [8] virtual machine and was specially developed for ARM processors. The KVM supports many different 16 and 32 bit micro processors which run at a minimum of 25MHz. Its memory demand is 160KB ROM and 32 KB RAM. The language complexity is specified by the CLDC 1.1 which also defines the supported API together with the MIDP.

The Squawk VM [7] was developed to run on the Sun SPOTs, developed by SUN Microsystems. It is nearly solely written in Java and does not need an operating system but comes with OS functions of its own. The target processor is the 32 Bit ARM-9 at a clock speed of 180MHz. Its memory demand is 149KB for the VM, 363KB for the VM suite, 158KB for the library suite in ROM. The Sun SPOTs have 512KB RAM from which about 20% are needed by the VM. The Java language is fully supported and the set of provided APIs are defined by the CLDC 1.1 extended by APIs for 802.15.4 radio communication and different low level hardware features.

The NanoVM [2][4] is a very limited VM which is distributed unter the GPL license and which runs e.g. on the Asuro roboters. The target platform is an ATmega8 or ATmega32 which both are 8 Bit microcontroller running at 8MHz clock frequency. The memory demand of this VM are 8KB ROM and less than 1KB RAM. It comes with a limited language support and only supports 15/31 Bit integer arithmetic, limited inheritance and a proprietary mechanism to support native code. There is no API present, only some native methods implemented in C.

The ParticleVM described in [6], is based on the NanoVM and is supported by the ParticleOS. Its memory demand is 45KB ROM and 0,5KB RAM. The target platform is a PIC18F6720 8 Bit microcontroller which runs at 20MHz. Its memory demand is 60KB ROM and 1,5KB RAM. The language support is the same as in case of the NanoVM, but the ParticleVM additionally supports interfaces. The API contains 20 classes with special functions for deployment and code migration via radio.

3. THE AMBICOMP VM

Like its small brother, the NanoVM, the new AmbiComp VM targets small microcontrollers such as the AVR 8-bit family, but provides more functionality than the NanoVM. The ACVM uses an external transcoding step, which transforms the original SUN Java opcodes and provides class loader functionality. This enables a very small footprint of the VM binary itself while supporting almost all SUN Java opcodes and a substantial part of the J2ME API. Moreover, the ACVM has been designed to support different target platforms with varying memory and processing capabilities. However, as a consequence, the entire bytecode that is needed by the ACVM must be passed through a transcoder instance before start up of a particular VM instance.

The tool chain for the ACVM starts with a regular *javac* compiler. It compiles a general albeit small Java program into one or more *.class*-Files. Then the transcoder takes this bytecode, transcodes it for the respective target platform, and links it, for example, statically to the plattform specific system libraries. Typically, only a small part of these libraries will be actually used. Thus the transcoder can eliminate a large portion of the library code.

The result of these transcoding process is a so-called *binary large object (BLOB)* file, which – in the case of static linking – contains the entire code that is needed to run the application on the target platform's ACVM. The BLOB is then transferred to the embedded device, for example, via Ethernet, USB, Bluetooth, or Zigbee depending on its respective networking capabilities. When the BLOB is fully loaded the ACVM starts to execute it.

Within the *AmbiComp* project, this tool chain will be fully integrated into the *Eclipse* software development environment, so that a programmer can directly deploy its code onto the target platform. In order for the transcoder to particularly transcode for respective target platform, the Eclipse user must select the desired target platform. The BLOB will then only be able to run on the according ACVM variant. Nevertheless, the Eclipse plugin will also have emulation capability so that the developers can check their program for compatibility with the respective target platform before deployment.

Unlike the NanoVM, the ACVM does not allow external users to provide native code directly. It comes however with several low level functions such as direct access to input and output pins of the microcontroller. Thereby, the ACVM – together with the system libraries – provides much of the functionality that is normally provided by the operating system: scheduling of threads in the VM, memory management and memory protection including garbage collection, and all kinds of IO functionality including the communication stacks. Hence, the ACVM does not need any further operating system.

4. DISTRIBUTED OPERATION OF ACVM

So far, we have described the ACVM as a stand-alone system. This impression is not quite true because the ACVM can access external objects. To this end, it makes use of the scalable source routing (SSR) protocol, which provides key based routing (KBR) for low-resource embedded nodes. A detailed description of SSR can be found in [1].

The ACVM uses SSR to route object access requests to remote objects. It can do so because KBR can address objects rather than nodes. Thus, it is easy to retrieve an object by its globally unique identifier. Unlike other distributed Java VMs this approach does not need any centralized server.

We will illustrate the use of this beneficial property with the following example: Consider a scenario where a temperature sensor shall send its measured values to a display device. Similar to a multi-threaded local system we use a sensor thread (cf. listing 1) and a display thread (cf. listing 2). Both are coupled via a singleton which stores the data that is published by the sensor (cf. listing 3). The display could regularly pull the published values from the data store, or it registers a listener with the data store in order to have the values pushed. To this end, the display implements the respective listener interface (cf. listing 4).

```java
package push.sensors;

import hardware.TemperatureSource;
import push.TemperatureDataStore;

public class TemperatureSensor {

  private TemperatureDataStore dataStore;
  private TemperatureSource sensorHardware;
  private String name;

  public TemperatureSensor(String name) {
    this.name = name;
    this.dataStore = TemperatureDataStore.
        getInstance();
    this.sensorHardware = new TemperatureSource();
  }

  public void run() throws InterruptedException {
    while (true) {
      Thread.sleep(500);
      System.out.println(”s” + name + ” pushed”);
      this.dataStore.store(
          this.sensorHardware.getValue()
      );
    }
  }
}
```

Listing 1: The sensor

```java
package push.actuators;

import hardware.Display;
import push.IDataListener;
import push.TemperatureDataStore;

public class TemperatureDisplay implements
     IDataListener {

  private Display actorHardware;
  private String name;

  public TemperatureDisplay(String name) {
    this.name = name;
    // create access to hardware
    this.actorHardware = new Display();
    // register ourself as listener to the data store
    TemperatureDataStore.getInstance().
        addDataListener(this);
  }

  public void notify(float value) {
    actorHardware.show(this.name + ”: ”+ value + ”C”);
  }
}
```

Listing 2: The display

```java
package push;

import java.util.ArrayList;
import java.util.List;

public class TemperatureDataStore {

  // we are a singelton
  private static TemperatureDataStore instance = null;
  // the actual data store
  private List<Float> temperatureValues;
  // listeners to be notified about new temperature
  private List<IDataListener> dataListeners;

  private TemperatureDataStore() {
    this.temperatureValues = new ArrayList<Float>();
    this.dataListeners =
        new ArrayList<IDataListener>();
  }

  /**
   * This class implements the singelton pattern.
   * @return The instance of this singelton.
   */
  public static TemperatureDataStore getInstance() {
    if (instance == null) {
      instance = new TemperatureDataStore();
    }
    return instance;
  }

  /**
   * Publish a value to the store.
   * @param value The value to be stored.
   */
  synchronized public void store(float value) {
    this.temperatureValues.add(value);
    notify(value); // notify listeners
  }

  /**
   * Notify all registered IDataListeners
   * about a new value.
   * @param value The new value.
   */
  private void notify(float value) {
    for (IDataListener listener : dataListeners) {
      listener.notify(value);
    }
  }

  /**
   * Adds a new IDataListener to the list of listeners.
   * @param listener The listener to be added.
   */
  public void addDataListener(IDataListener listener) {
    this.dataListeners.add(listener);
  }
}
```

Listing 3: The data store

```
package push;

public interface IDataListener {
  public void notify(float value);
}
```

Listing 4: The listener

Since the data store is a singleton, only one such instance will be available (in the local domain). Accordingly, the respective class member will be remote for the nodes except for the one that holds this member. SSR will find this node based on its globally unique ID. Such an ID is obtained by combining a BLOB-local ID with the hash of the BLOB that defines the member.

At the current stage of the project, this mechanism is only about to be available in practice. Moreover, not all design decissions have been made up to now. Let us briefly sketch some of the respective problems.

For example, the concept of administrative domains has only been phrased roughly. It addresses the fact that depending on the application, singleton objects should not be global but somehow local. In the described scenario, the temperature sensor and the display should belong to the same household or the same machine. Thus, the ACVM needs to be equipped with a means to let the application developers decided which singleton domain they want to have for a particular aspect of their application. We envisage to support this via an API, but it is still unclear which granularity of control this API should provide.

Another open issue that has not been decided yet is the question of how to treat synchronized remote objects. On a local machine synchronized objects are rather simple because deadlocks can be considered a bug in the application. In a distributed environment deadlocks can also be caused by failing nodes or lost communication links. At the time being, it is unclear to what extend the application programmer needs to be confronted with these issues.

Furthermore, we are still considering various aspects of object and code migration. For example, BLOBs cannot only be linked statically, but also dynamically so that other BLOBs are loaded on demand. This can again be done via SSR's KBR semantic, for example, with exploiting caching of BLOBs in nodes that have enough memory. This will enable each node to load and execute software on demand whenever it is needed. Moreover, this feature also allows to trade several remote object accesses (ROA) for potentially one remote method invokation (RMI). Albeit it is rather easy to implement RMI on top of SSR, it is yet unclear how a simple heuristic can help the ACVM decide between ROA and RMI.

5. SUMMARY

In this paper we have described ongoing work in the AmbiComp project: We have briefly presented the AmbiComp Virtual Machine (ACVM) that executes BLOBs that an external transcoder creates from Java *.class*-files; we have also described how the ACVM together with the scalable source routing (SSR) protocol manages remote object access; and we have illustrated the use of this idea with a simple temperatur sensor and display example.

Yet, since this is still very early work, we have not addressed all the open issues in this field. In essence, the ACVM can provide a powerful means to hide the distributedness of a sensor-actor-system. But in practice the application developers need a way to craft their distributed system. To this end they need to decide, for example, on the administrative domain in which an application may exchange data, or on the semantics of synchronization in case of failing nodes. We hope therefore that this paper stimulates the discussion about these questions.

6. REFERENCES

[1] T. Fuhrmann. Scalable routing for networked sensors and actuators. In *Proceedings of the Second Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, Sept. 2005.

[2] T. Fuhrmann and T. Harbaum. A platform for lab exercises in sensor networks. Technical report, Z?rich, Switzerland, Mar. 23–24 2005.

[3] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification.* Addison-Wesley Professional, third edition, July 2005.

[4] T. Harbaum. The NanoVM - Java for the AVR, 2005. `http://www.harbaum.org/till/nanovm`.

[5] B. Haumacher, T. Moschny, J. Reuter, and W. F. Tichy. Transparent distributed threads for java. page 147, Nov. 12 2003.

[6] T. Riedel, A. Arnold, and C. Decker. An OO Approch to Sensor Programming. In *EWSN 2007 - Adjunct poster proceedings*, number PDS-2007-001, pages 39–40, Delft, The Netherlands, Jan. 29-31 2007.

[7] N. Shaylor, D. N. Simon, and W. R. Bush. A java virtual machine architecture for very small devices. In *ACM SIGPLAN conference on Language, compiler, and tool for embedded systems (LCTES)*, pages 34–41, New York, NY, USA, 2003. ACM Press.

[8] Sun microsystems. *CLDC HotSpot Implementation Virtual Machine*, Feb. 2005. `http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf`, accessed on 2007-05-21.

[9] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.