JSense - Prototyping Sensor-Based, Location-Aware Applications in Java

Silvia Santini, Robert Adelmann, Marc Langheinrich, Georg Schätti, and Steven Fluck

Institute for Pervasive Computing
ETH Zurich, 8092 Zurich, Switzerland
{santini, adelmann, langheinrich}@inf.ethz.ch
{schaetti, stfluck}@student.ethz.ch

Abstract. Creating applications based on data from individual sensor nodes is typically a two-tiered process: Firstly, a (potentially large) number of sensor nodes is deployed in order to gather comprehensive datasets. After analyzing the collected data, algorithms are then installed on the individual nodes and iteratively fine-tuned using a collect-and-analyze procedure. This approach is not only time consuming, but also prone to errors: the two separate steps of data collection and data analysis complicate algorithm development; the absence of programming abstractions in embedded systems programming languages often introduces hard-to-detect runtime errors; and the lack of modern integrated development environments (IDEs) does not allow for quick trial-and-error prototyping. To mitigate those effects, we have developed JSense – a hardware abstraction layer for modern sensor nodes that allows for Java-based access to all sensor and actuator controls. It supports an IDE-based centralized development cycle with real-time debugging of a particular sensor environment, as well as the use of not-yet-available sensor and actuator hardware on each node, such as positioning information. Using JSense, designers of sensor-based environments can quickly try out a combination of situations and observe in real-time the data collection processes of their nodes, while programmers are able to prototype applications in their favorite Java-IDE in a hardware independent fashion, even taking into account not-yet-deployed node hardware.

1 Introduction

Sensor-based applications form the backbone of the field commonly known as ubiquitous computing or ambient intelligence. Without sensors, popular visions of "an environment recognising and responding to the presence of individuals in an invisible way" [1] will hardly be possible. Wireless sensor nodes such as Berkeley's MICA mote [2] or ETH Zurich's BTnodes [3] are envisioned to become both considerably smaller and cheaper, thus eventually allowing "an end-user buying a collection of sensor nodes and gateways, powering them up, and sprinkling them – literally and figuratively – within an environment"[4]. However, even if such motes would cost cents, not dollars, and even if they would be small enough to be "sprinkled" into our environment, it is far from clear how an end-user would actually go about *programming* these systems to do what she or he wants. This is because wireless sensor nodes today largely remain within the realm of embedded systems programming – typically requiring: intimate knowledge of the hardware being used (in order to correctly read out sensor values and communicate them wirelessly); the means to perform *in-system-programming*[1] (in order to upload new programs); experience with embedded debugging tools; and proficiency in low-level programming languages such as assembler or C.

Even if a user would fit this description, the development of a particular application would most likely still be tedious: Uploading programs to dozens of nodes, as well as debugging them individually, is a time consuming process. Moreover, due to the lack of abstractions in typical embedded system programming languages such as C or assembler, the likelihood for non-trivial runtime errors such as invalid pointers, improper stack management, or memory leaks significantly increases. Last not least, the lack of direct control over individual nodes typically leads to a non-integrated (hence suboptimal) design process, where a first stage collects a multitude of sensory information (often spatially and temporally over- or undersampled) while a second stage is used to separately design an application on top of such datasets.

In this paper, we describe a Java-based network interface to wireless sensor nodes that significantly simplifies the design and development cycle for wireless sensor node applications. Our system, called JSense, works by deploying a small hardware-specific access and control layer on each individual sensor node. One or more gateway nodes can than be controlled via a set of common Java APIs to read out or send commands to each individual node, thus providing programmers with a Java-based direct access interface to their sensors. JSense's Java-based approach not only supports an "armchair" development-cycle (i.e., programmers can collect and analyze data, and eventually reprogram their sensor-based application from within their favorite Java-IDE), but also improves code portability and quality through the use of Java's high-level programming abstractions.

In addition to providing an easy-to-use sensor and actuator interface, JSense's hardware abstraction layer also supports the inclusion of *external sensors*, i.e., sensory information attributed to an individual node yet not collected by its own sensors. Examples for such an external sensor would be temperature information from an infrared camera picture, or a node's location determined with the help of an external positioning system. External sensors further simplify the development of sensor-based applications, as they not only allow for the inclusion of yet-to-be-released hardware (e.g., a GPS-enabled BTnode) but also transparently support the use of multiple sensor technologies (e.g., UWB, GPS, ultrasound) for performance comparison or across different environments (e.g., indoor vs. outdoor).

[1] In-system-programming describes the process of directly storing a program on an embedded microchip's flash memory, e.g., using a serial cable and a corresponding hardware programming device.

After briefly describing the main challenges related to the development of sensor-based applications and our concrete hardware setup in sections 2 and 3, we will present the JSense architecture and its implementation details in section 4. Section 5 will contrast our approach to existing development environments, while section 6 concludes with a summary and an outlook on future work.

2 Developing Sensor-Based Applications

Sensor-based applications use measurements from real-world sensor deployments to offer a particular service, e.g., a motion detection measurement might be used to automatically switch on the lights when a person enters a room. Using multiple sensors (both in terms of numbers and in terms of sensor types) can often significantly improve such an application, e.g., a co-located light sensor might help our motion sensor from above decide whether there is actually a need for lighting during daytime. Developing such a simple application on a typical sensor platform would require the developer to learn not only the basics of embedded systems programming (i.e., hardware-near programming languages such as C or assembler, microcontroller memory management, and in-system programming), but also the accompanying APIs of the employed sensors and may be even the platform's radio module. It is widely recognized that the absence of a common, extendible, and easy-to-use programming interface for accessing real-world sensors still represents a major burden for a rapid prototyping of sensor-based applications [4–7].

A typical approach to simplify the initial development of such applications is the use of a simulator: instead of having to deal with the intricacies of actual sensor hardware, the developer can use abstract sensor nodes with simulated sensor readings that can quickly be programmed and debugged. While simulations are a powerful tool for evaluating preliminary design and configuration choices, they often fail to capture the complex, real-time interactions between the application software and the faulty-prone, physical sensors. In order to build reliable sensor-based applications, real-world deployments must be an integral part of the development cycle [4, 5].

So what do we need to support application developers in deploying their ideas onto actual sensor platforms, so that they can quickly try out a variety of approaches to provide higher-level services from low-level sensor data? Based on our own experiences in teaching sensor-based application development to students, as well as by reviewing a number of state-of-the-art development environments [4, 5, 7–9], we have distilled five core requirements:

1. *Hardware Abstraction:* Instead of requiring developers to learn about embedded systems programming (which entails both hardware and software concepts), they should be able to simply query a particular sensor on a particular node, or set a node's actuator (e.g., an LED), through a high-level, unified API.
2. *Integrated Design Process:* In order to avoid a suboptimal, decoupled design process (i.e., separate stages for data collection and algorithm design/testing),

the framework should allow near real-time gathering and analysis of collected sensor readings. This allows developers to receive direct feedback on algorithm design under controllable real-world conditions.

3. *Centralized Programming Environment:* While sensor-based applications will ultimately need to be distributed onto individual sensor nodes, the process of programming and debugging sensor nodes one by one is time consuming and error prone. Instead, developers can greatly benefit from a centralized programming environment that lets them (virtually) upload new program versions in an instant and quickly observe the results.
4. *High-Level Programming Language:* Embedded microcontrollers are typically programmed using the C language, as it allows for a direct control of the individual IC registers and flags. In contrast to higher-level languages such as Ada or Java, however, low-level languages such as C or assembler fail to support reliability and maintainability, nor do they try to address the compile-time detection of errors [10, 11]. Providing developers with a Java-interface would thus not only improve the code quality, but also lower the barrier of entry, as an increasing number of universities, colleges, and secondary schools have long since adopted Java as the programming language for their introductory computer science courses [12].
5. *Location Information:* In most sensor-based applications scenarios – from large scale environmental monitoring [13] to smart-buildings applications [14] – reported measurements are often useless if they are not accompanied by a corresponding (absolute or relative) sensor position. In the smart room example cited above, the controlled light switch must obviously be the one that is co-located with the light and motion sensors. An application thus greatly benefits from having direct access to the geometric or symbolic [15] coordinates of the sensors within the actual deployment area.

With these five requirements in mind, we have developed JSense, a Java-based direct access and control interface to common sensor platforms. Like a number of similar rapid prototyping environments for sensor nodes (which we will discuss in detail in section 5 below), JSense aims at speeding up the prototyping phase of sensor-based applications by providing a high-level API for programming heterogenous sensor platforms. As we will describe in details in section 4, the JSense API provides a set of basic programming primitives for activating and deactivating sensors, importing sensor data streams, and eventually set parameters like sampling frequencies or actuators states. Unlike many other approaches, however, JSense provides an application developer with an easy-to-use *Java* interface, which – due to its popularity as a programming language in universities and colleges [12] – offers a significant potential to use JSense specifically as an educational tool, e.g., in tutorials accompanying courses in embedded systems, sensor networks, or ubiquitous computing. The adoption of Java as a programming language also facilitates the use of external data processing tools (e.g., Matlab) that already offer Java bindings [16], thus further improving applications development.

Secondly, JSense explicitly supports the use of *location information* on the sensor node. In many experimental settings, location data is typically retrieved from an external database, where the position of the single sensing devices is registered during deployment. This solution not only seriously limits reconfigurability, but also makes the system prone to inaccuracies, as sensors could be moved and thus would invalidate the information in the database. However, as only few of today's popular sensor platforms supports location sensing, JSense provides the concept of *external sensors* in order to allow developers to seamlessly use third-party positioning systems (e.g., GPS or Cricket, but also optical systems based on fiducial markers) as if the positioning data would be generated by the actual sensor platform.

Before describing the JSense architecture in detail (and in particular its Java-interface and location data support), we will briefly describe our particular hardware setup, i.e., the sensor platform and location system that we have used to develop our initial prototype of JSense: The Tmote Sky sensor platform and the Ubisense location system.

3 JSense Hardware Setup

In the context of this work we use the term *sensor platforms* to refer to a device endowed with one or more sensors and/or actuators, some computational capabilities, and means for wireless communication. A sensor platform can thus consist of a single, stand-alone temperature or orientation sensor [17], or be a more complex device that offers a range of sensing capabilities [3, 18, 19].

One popular sensor platform family – the Berkeley MICA motes – is based on work originally done at UC Berkeley and Intel Research [20]. Its latest generation is the "Tmote Sky" sensor mote, which is developed by Moteiv, a UC Berkeley spin-off company [21]. It features an IEEE 802.15.4 compliant radio transceiver; built-in temperature, light and humidity sensors; and three LEDs. The Tmote Sky platform can be programmed using "TinyOS", a component-based, open-source operating system widely used for research in wireless embedded systems [22, 23]. Applications on top of TinyOS are developed through composition of independent modules, which must be written using "NesC", an extended dialect of the C programming language [24]. TinyOS has already been ported to a large number of hardware platforms and is therefore widely used within the sensor network community for research and development. Due to the popularity of both TinyOS and the Berkeley motes family, we have started our initial development of JSense using the Tmote Sky sensor platform. While programming the Tmote Sky motes is well supported by the (mote-specific) NesC programming language, it is still hampered by the general drawbacks of C-programming, as well as the difficulties of learning the needed TinyOS programming paradigm.

Although some commercially available platforms also integrate positioning devices, such as GPS receivers [25], [19] wireless sensor platforms typically do not feature any integrated positioning devices, due to their high costs and significant energy requirements. However, as we pointed out in the previous section, the ability to retrieve position information is often crucial in order to validate sensor-based application design and system configuration. We have thus incorporated the ability to include *external sensors* when modeling a sensor platform in JSense, i.e., sensors that operate independently of the actual sensor platform used, but which can be correlated with individual nodes such that they form a single, virtual node. We use this mechanism to incorporate location information from an external positioning system directly into the representation of every single Tmote Sky, making it appear as if this sensor platform would already be equipped with such a positioning technology.

The particular location system we use is based on ultra wide band (UWB) technology, which promises energy-efficient, accurate positioning in both indoor and outdoor settings. Systems that use this technology for getting 3-dimensional indoor positioning information are already commercial available, e.g., the Ubisense system [26]. We have installed a set of Ubisense sensors in our student lab, which are able to report the geometrical coordinates of corresponding Ubisense *tags* (i.e., a small UWB radio beacon) with an accuracy of up to 15 cm. We thus enhanced the sensing capabilities of the Tmote Sky platform by attaching a Ubisense tag to the sensor node, and then provided the necessary Java software interface to our JSense architecture in order to integrate positioning information into the regular Tmose Sky sensor readings (i.e., temperature, light, and humidity). JSense in effect allows application developers to use this compound platform as a unique, homogenous entity.

The next section will explain in more detail how our JSense architecture combines the measurements collected by the Tmote Sky sensing devices with the position information computed by the Ubisense system.

4 The JSense Architecture

The first core component of JSense's two-tiered architecture sketched in figure 1, is implemented through so-called *Virtual Platforms*. As we will detail in section 4.2 Virtual Platforms (VP) are software entities accessible through a standardized Java API that virtually bind together a compound of different sensor platforms. In our prototypical implementation, for example, a VP combines a Tmote Sky sensor node and the correspondent affixed Ubisense tag in a unique virtual sensor platform.

JSense's second core component consists in a lightweight, platform-specific, hardware access and control layer, that shields the application developer from the nasty hardware-specific details of the sensor platforms. This framework, described in more detail in section 4.1 below, enables remote access to the sensing devices available on the physical hardware platforms and is easily extendible due to its component-based architecture.

4.1 Local Command Execution: The Hardware Abstraction Layer

For enabling ease of access to the sensing devices of a sensor platform, JSense provides a hardware-specific access and control layer, the so-called Hardware

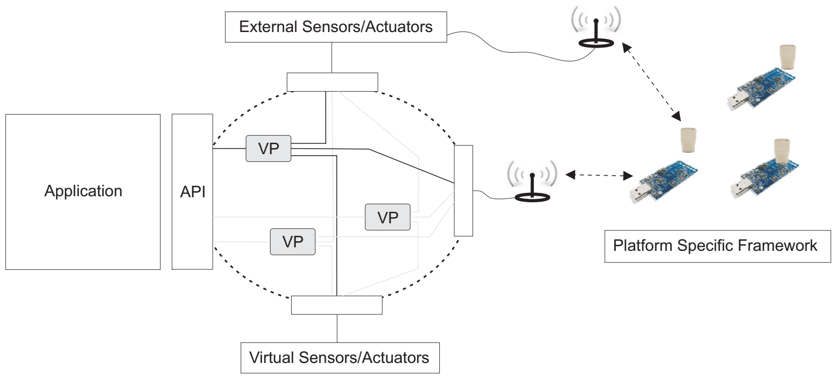

External Sensors/Actuators
Application
API
VP
VP
VP
Platform Specific Framework
Virtual Sensors/Actuators

Fig. 1. The JSense architecture.

Abstraction Layer (HAL). The HAL runs on the specific platform it has been written for and uses a pluggable-module approach, where each sensor or actuator is accessed through an individual driver module. The HAL is not only responsible for scheduling and executing hardware access (using the sensor specific modules) but also of managing the received access requests and returning the adequate sensor responses. The existence of this abstraction layer allows the application developer to issue remote access requests that will be received and executed on the target platform, while the correspondent sensing results will be back-forwarded to the remote system, e.g., a desktop PC.

Figure 2 shows the implementation of JSense's Hardware Abstraction Layer on the Tmote Sky sensor platform. Its lightweight, easy-to-extend implementation is based on three main components. The *SensorComm* component manages the communication with the remote system: it is responsible for receiving remotely issued commands and forwarding them (after adequate unmarshalling) to the *SensorLogic* component. Commands typically have the form "get current value of sensor X" or "perform X on actuator Y". In order to reduce, when possible, the communication overhead, commands that need to be executed periodically can be issued by the remote system as periodic access requests. This requests will perform sensor/actuator access with the specified frequency and for the desired time frame.

The duty of scheduling sensor/actuator access is taken over by the *SensorLogic* component, which handles commands coming from the *SensorComm* component and either executes them immediately or provides the adequate scheduling for periodic execution, as illustrated in figure 3. When a command needs to be executed (thus, a sensor need to be read or an actuator to be set), the *SensorLogic* component activates the correspondent sensor- or actuator-specific module, which is responsible for the actual physical access to the hardware device. The compound of these modules constitutes the third logic component of our HAL. Please note that enhancing the sensor platform with additional sensors and/or actuators, just requires adding to the JSense'S framework the correspon-

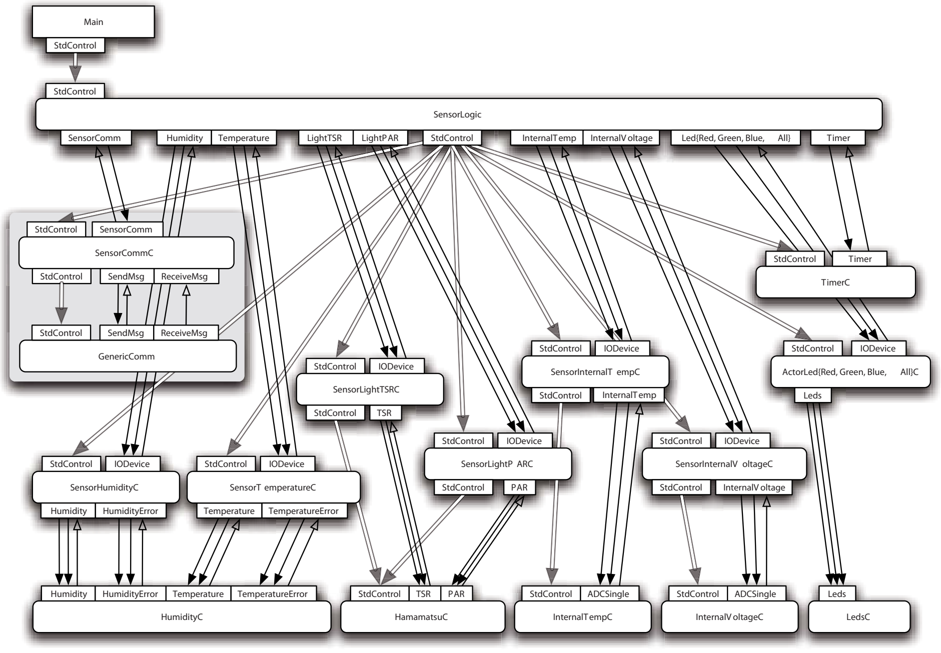


Fig. 2. Component model of the JSense's Hardware Abstraction Layer for the Tmote Sky sensor platform.

dent sensor-/actuator-specific physical access modules and registering them to the *SensorLogic* component.

Access request responses sent through the *SensorComm* component are received by a Tmote Sky node connected to a remote system (e.g., a desktop PC) through a USB serial port. This node acts as a gateway between the sensor deployment and the desktop PC by transferring received radio packets to the *SerialForwarder*, a Java application that is part of the TinyOS tool-chain. The *SerialForwarder* listens for TinyOS packets on a serial port and forwards them over a local TCP network socket, thus allowing more than one application to send and receive packets to and from the "gateway" Tmote Sky.

The current JSense's HAL implementation for the Tmote Sky sensor platform, allows for an easy and efficient access to the physical sensors and actuators, generating a 63kB footprint on the 1MB flash memory of the Tmote Sky platform. This footprint includes both the operating system proprietary modules and the JSense's HAL components for accessing the platform's built-in sensors and actuators.

4.2 Remote Access: JSense's Virtual Platforms

As mentioned earlier in this section, a Virtual Platform is a software entity that virtually binds together a compound of different real sensor platforms. Figure 4

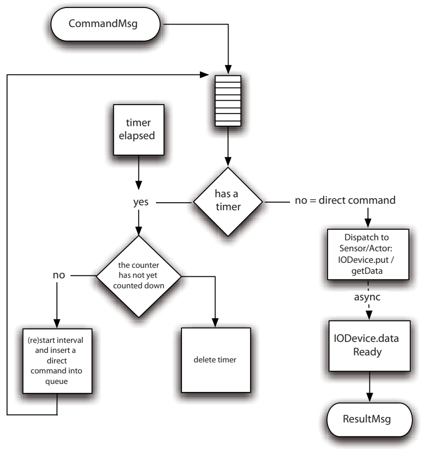

CommandMsg
timer elapsed
yes
has a timer
no = direct command
Dispatch to Sensor/Actor: IODevice.put / getData
async
no
the counter has not yet counted down
(re)start interval and insert a direct command into queue
delete timer
IODevice.data Ready
ResultMsg

Fig. 3. Scheduling of sensors/actuators access as implemented by the *SensorLogic* component.

shows that this virtual entity typically integrates one or more target platforms and eventually external and virtual sensors. We refer to a *target platform* as a sensor platform that runs a JSense's Hardware Abstraction Layer, for instance, the Tmote Sky sensor node.

The possibility to add external sensors to a target platform represents a powerful feature of the JSense system, as it allows for incorporating functioning sensor hardware that is not (yet) integrated into the target platform. We assume that external sensors can communicate independently over a wireless channel, or at least can be assessed from the outside (e.g., using a camera-picture or a microphone array). For instance, a location sensor can be "attached" to a target platform in order to enable prototyping of location-aware, sensor-based applications. Due to the often significant power, cost and size requirements of location sensors such as GPS receivers or UWB tags, these devices are typically not integrated on common sensor node platforms (such as MICA-motes or BTnodes). However, using the VP abstraction we can easily integrate a target platform with an external GPS module or a UWB-tag, allowing the application developer to access these devices as they were built-in sensors of the target platform, thus completely hiding the existence of an external positioning system from our development cycle. This not only provides a unified access and control framework to heterogenous sensor platforms, but also support the simultaneous use of multiple technologies, thus offering a precious tool for e.g., performance comparisons.

Target platforms can be integrated not only with external, but also with so-called *virtual* sensors. These represent devices that either cannot be purchased (e.g, for cost or availability reasons) or even do not yet physically exist. In both cases, the virtual sensor simulates the existence of a real sensor. In effect, using the concept of virtual sensors, we can develop sensor-based applications also integrating yet-to-be-released or yet-to-be-operational sensing devices, significantly shortening the development cycle.

Once sensing devices have been associated to a certain VP, JSense provides a transparent access to the physical hardware through a standardized Java API. The development of an application that makes use of the deployed sensor hardware is therefore reduced to the compilation of a standard Java program that access the correspondent VPs. For the application programmer, a VP is represented by a Java object that provides methods for a direct access to its sensors and actuators.

The VP abstraction supports the development and implementation of both *distributed* Java applications (i.e., each virtual node is a separate thread with individual code) and *centralized* applications (i.e., virtual nodes are simply objects that can be queried and accessed from a central program). This allows developers a gentle learning curve into sensor-based application development, as one can quickly prototype a centralized application, and then gradually distribute it onto individual nodes.

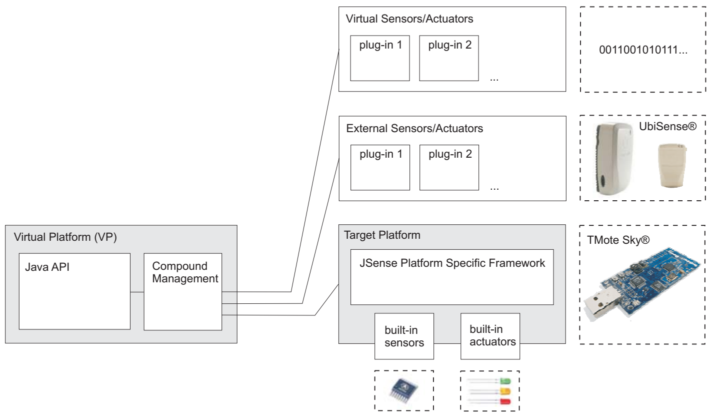


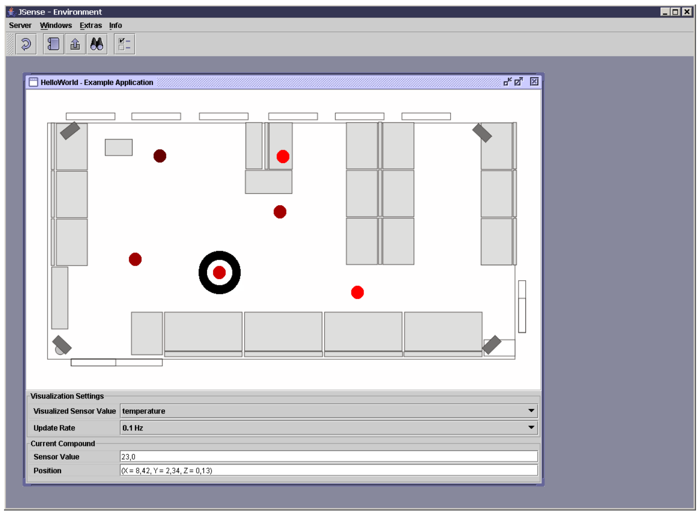
Fig. 4. A JSense Virtual Platform as a compound of a target platform, external and virtual sensors.

We would like to point out that integrating new external sensor and actuator systems into JSense requires writing the appropriate Java bindings and register them as plug-ins components using methods provided by the JSense Java API. For instance, the Ubisense UWB localization system has been integrated in JSense by implementing a software layer that provides access to the Ubisense native C++ libraries through the functionalities provided by Sun's Java Native Interface[2].

[2] See `www.java.sun.com/j2se/1.5.0/docs/guide/jni/index.html`.

4.3 JSense Enabled Systems

The JSense system is in general very well suited for centralized implementation and testing of applications requiring on-line processing of real-world sensor data. For this kind of applications, the enrichment of sensor readings with the correspondent location information represents in many cases a mandatory feature. For this reason, the JSense system provides the Ubisense UWB-based localization system as an external sensor integrating the Tmote Sky sensor node platform.



Visualization Settings

Visualized Sensor Value: temperature

Update Rate: 0.1 Hz

Current Compound

Sensor Value: 23,0

Position: (X = 8,42, Y = 2,34, Z = 0,13)

Fig. 5. Application development using JSense: the graphical interface.

A set of 6, Ubisense-integrated, Tmote Sky sensor nodes deployed in our student lab, provided the hardware setting for prototyping our first, 'hello-world"-style JSense-based application. The developed application aimed at collecting temperature readings (with an update rate of 0.1 Hz) from the deployed nodes. The temperature data and the correspondent position information from the Ubisense system could be visualized through a simple graphical interface (see figure 5) and were logged in a protocol file for off-line inspection. Figure 6 shows the simple Java program that it was necessary to write for implementing the described application. As it can be seen from the code, accessing sensor or actuators on the target platform or adding an external sensor to a VP, require just simple Java method calls.

Thus, using JSense a first prototype of a sensor-based, location-aware application can be written and tested in Java, having all the benefits of unrestricted system resources and integrated development environments. On this level, the logical correctness of the approach can be tested and the general functioning in combination with real sensor data can be assured – with basic Java knowledge being the only required skill for the application developer.

```
import jSense.*;
import jSense.Sensors.*;
import jSense.Actuators.*;
import jSense.ExternalSystems.*;
import tools.protocol.*;

public class JSenseDemo implements VPListener {

  private VirtualPlatform vp;

  TemperatureSensor tempSensor;
  LEDActuator ledActuator;
  LocationSensor = locationSensor;

  public JSenseDemo() {
    // create a new virtual platform associated to a TMoteSky node (logical adress 1):
    vp = new VirtualPlatform(new TMoteSkyPlatform(1));

    // add components: (UbiSense tag number 016-000-003-180)
    ExternalSystem localizationSensor = new UWBLocationSensor("016-000-003-180");
    vp.add(localizationSensor);

    // create shortcuts to the used sensors and acuators:
    tempSensor=(TemperatureSensor)((TMoteSkyPlatform)vp.getPlatform()).getTempSensor();
    ledActuator=(LEDActuator)((TMoteSkyPlatform)vp.getPlatform()).getLEDActuator();
    locationSensor=(LocationSensor)(UWBLocationSensor)vp.getExternalSystems().elementAt(0);

    // initialize the protocol system:
    Protocol protocol = new Protocol("c:/temperature_readings.txt");
    protocol.setTimeStampsEnabled(true);
  }

  public void directAccessExample() {
    // get the current temperature value:
    float temperature = tempSensor.getTemperature();
    // turn the first LED on:
    ledActuator.setLED(0, true);
  }


  public void eventBasedAccessExample() {
    vp.subscribe(this);
    // get 100 temperature readings every 1000 ms, starting now (0 ms):
    vp.issueTask(new SensorReadingTask(tempSensor,0,100,1000));
  }


  /** This method is called if the VP receives a message regarding an issued task*/
  public void resultReceived(Result result) {
    if (result.getSensor() instanceof TemperatureSensor) {
      TemperatureSensor tempSensor = (TemperatureSensor) result.getSensor();
      // write the obtained values into the protocol file, incuding the current system time:
      Protocol.println("temperature:"+tempSensor.getTemperature()+
                       "at position:"+locationSensor.getPosition());
    }
  }

  public static void main(String[] args) {
    JSenseDemo demo = new JSenseDemo();
    demo.directAccessExample();
    demo.eventBasedAccessExample();
  }
}
```

Fig. 6. JSense enabled, sensor-based application written as a standard Java program. Using the JSense package, sensors and actuators can be accessed through simple methods calls.

5 Related Work

Due to the growing interest in wireless sensor applications, researcher have put considerable effort in the development of tools for allowing easy development and prototyping of wireless, sensor-based systems. For instance, a number of interesting approaches like TASK [4], EES [5] and SNACK [7], have been proposed within the sensor networks research community.

The "Tiny Application Sensor Kit" (TASK) provides sensor networks end-users with a set of graphical tools for facilitating deployment and inspection of environmental monitoring systems. It has been successfully used in real-world experiments and complies with the need of end-users providing, among others, ease of software installation, simple deployment tools for sensor placement, re-configurability and ease of integration with data analysis tools. TASK has been specifically designed for low data rate environmental monitoring applications and relies on the TinyDB querying processor [27]. The data collected in the context of TASK-enabled real-world experiments have been made publicly available [28] and served as benchmark for evaluating data processing technique for sensor networks [29, 30]. Unlike JSense, TASK does not support easy extendability of the sensor platform and is specifically designed for inexpert computer users, rather than for application developers that are familiar with high-level programming languages like Java. The same considerations apply for the Extensible Sensing System (EES), that has been used to collect micro-climate data from large scale (> 100 nodes) outdoor deployments. Unlike TASK, the sensor nodes in ESS are heterogeneous and run a proprietary *Data Sampling Engine* rather than TinyDB.

One of the approaches we retain to be more closely related to our work is the "Sensor Network Construction Kit" (SNACK). SNACK is a NesC-based system that aims at providing a set of efficient, high-level service libraries to support sensor network application development. However, while SNACK uses a proprietary component composition language that the application developer must eventually learn from scratch, JSense applications can be written using standard Java programming.

Other interesting systems, which are to some respect related to our work are Marionette [8] and IrisNet [9]. Marionette is a tool for interactive sensor network application development. It provides remote access to wireless embedded devices through a Phyton-based client and allows to monitor and change at run-time the state of a sensor node. The Marionette architecture promises to become a powerful tool for developing and debugging sensor network applications, but it address expert programmers that can understand and properly manage NesC modules, rather than Java developers, as JSense does. The IrisNet architecture aims providing the missing software components necessary to support realization of a *worldwide sensor web*, in which users can access through the internet a plethora of distributed, remote sensor systems. IrisNet holds distributed databases to store sensor data across the network and uses XML to query these databases. Sensors send their data to a so-called *Sensing Agents* (SA) that pre-processes the data and updates the sensor database distributed throughout the network. IrisNet's main scope is to provide a mean for a unified access interface to real-world sensor, but while JSense mainly access local sensors, reachable through a one-hop, wireless communication channel, IrisNet uses the world wide web infrastructure for accessing remotely located sensors.

Researchers at Sun Microsystems Laboratories very recently announced a project that aims at realizing a sensor development kit called Sun SPOT (Small Programmable Object Technology) [6]. A Sun SPOT platform is based on a 32 bit ARM CPU and an 11 channel 2.4 GHz wireless transceiver and is equipped with temperature, light and motion sensors. The Sun SPOT system does not need to rely on an operating system like TinyOS: it features the so-called "Squawk VM", a Java Virtual Machine mainly written in Java (only the interpreter and the garbage collector are written in C) that acts as an operating system running "on the bare metal" of the hardware platform. Applications building upon the Sun SPOT platform can be completely written in Java, using the native libraries of the Squawk Java VM. Since it is Java-based, the Sun SPOT will be easily integrable in our JSense system, and will thus constitute an additional sensor platform to experiment with. However, even if the the Sun SPOT technology is envisioned to enable rapid prototyping of some wireless sensor-based applications, it is still unclear if and how this technology would be mature enough to be extended to other hardware platforms. For instance, consider that while the Tmote Sky platform features $10kB$ of RAM and $1MB$ of flash memory, the Squawk VM has a $80kB$ footprint on the RAM and additional $270kB$ of libraries need to be loaded on the flash memory. Moreover, while considerable effort has been spent in the wireless sensor community to minimize cost of sensor platforms, a Sun SPOT development kit containing two wireless sensors, a base station and the software development tools will be available for about \$500. Three Tmote Sky can be purchased with less than the half of this money.

6 Conclusions

Today's means of designing, developing, and deploying sensor-based applications lack in flexibility, reliability, and convenience, thus seriously hampering sensor-based application development [4]. With JSense, we have presented a rapid prototyping environment for wireless sensor-based applications that offers developers not only a convenient Java interface for direct access and control of individual sensor nodes, but which also seamlessly integrates external sensor information, such as positioning data, in order to better design and inspect location-dependent sensor applications.

The support for Java programming puts sensor-based application development within the reach of millions of Java programmers and is especially relevant in the educational domain, where Java is often preferred over languages like C or C++ for introductory programming classes [12]. The seamless inclusion of location information as part of a Virtual Platform offers the promise of a much faster and more exact design cycle, as sensor readings can be directly correlated with a sensing position, alleviating the need for manual location measurements or fixed sensor locations.

JSense represents a first step towards a much simpler access to sensor-based applications, yet much needs to be done before we can use JSense to develop real-world applications. In particular, we kept the JSense Java API very simple in order to get a first, working prototypical implementation of the system. On the basis of this first experience, we are currently redesigning the sensor programming interface in order to allow more powerful querying primitives than single-value or periodic sensor queries. An almost trivial extension, for example, is to include support for spatial queries, i.e., based on a nodes location, as JSense's virtual platforms already support positioning information "natively".

Obviously, JSense will also benefit from extending it to run on a larger number of sensor platforms, e.g., the BTnodes. We are currently planning to release JSense as an open-source project in order to simplify the addition of different hardware. By using JSense in a number of student projects within our lab, we also hope to gain more insights into the practical uses (and shortcomings) of our system.

References

1. Ahola, J.: Ambient intelligence. ERCIM News (47) (2001) 8
2. Estrin, D., Culler, D., Pister, K., Sukhatme, G.: Connecting the physical world with pervasive networks. Pervasive Computing **1**(1) (2002) 59–69
3. Beutel, J., Kasten, O., Mattern, F., Römer, K., Siegemund, F., Thiele, L.: Prototyping wireless sensor network applications with BTnodes. In: 1st European Workshop on Wireless Sensor Networks (EWSN). Number 2920 in LNCS, Berlin, Germany, Springer-Verlag (2004) 323–338
4. Buonadonna, P., Gay, D., Hellerstein, J.M., Hong, W., Madden, S.: Task: Sensor network in a box. In: Proceedings of the Second IEEE European Workshop on Wireless Sensor Networks and Applications (EWSN 2005), Istanbul. Turkey (2005)
5. Guy, R., Greenstein, B., Hicks, J., Kapur, R., Ramanathan, N., Schoellhammer, T., Stathopoulos, T., Weeks, K., Chang, K., Girod, L., Estrin, D.: Experiences with the extensible sensing system ess. Technical Report 01-310-825-3127, UCLA Center for Embedded Network Sensing (2006)
6. Sun Microsystems Laboratories: The Sun SPOT Project. (Project Website: `www.sunspotworld.com`)
7. Greenstein, B., Kohler, E., Estrin, D.: A sensor network application construction kit (snack). In: Proceedings of the 2nd Intl. Conf. on Embedded Networked Sensor Systems (SenSys'04), Baltimore, Maryland, USA, ACM Press (2004)
8. Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., , Culler, D.: Marionette: Using rpc for interactive development and debugging of wireless embedded networks. In: Proceedings of the Fifth Intl Conf. on Information Processing in Sensor Networks (IPSN'06), Nashville, USA., ACM (2006)
9. Gibbons, P., Karp, B., Ke, Y., Nath, S., Seshan, S.: Irisnet: An architecture for a worldwide sensor web. IEEE Pervasive Computing **2**(4) (2003) 22–33
10. Wheeler, D.A.: Ada, c, c++, and java vs. the steelman. Ada Letters **XVII**(4) (1997) 88–112
11. Martin, P.: Java, the good, the bad, and the ugly. SIGPLAN Notices (1998)

12. Roberts, E.: Resources to support the use of Java in introductory computer science. In: SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education, New York, NY, USA, ACM Press (2004) 233–234
13. Szewczyk, R., Mainwaring, A., Polastre, J., Culler, D.: An analysis of a large scale habitat monitoring application. In: Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys), Baltimore, Maryland, USA (2004)
14. Holmquist, L., Gellersen, H., Kortuem, G., Antifakos, S., Michahelles, F., Schiele, B., Beigl, M., Maze, R.: Building intelligent environments with smart-its. Computer Graphics and Applications, IEEE **24**(1) (2004) 56– 64
15. Becker, C., Dürr, F.: On location models for ubiquitous computing. Personal and Ubiquitous Computing **9**(1) (2005) 20–31
16. Whitehouse, K.: Matlabcontrol.java. (Project Website: `www.cs.berkeley.edu/~kamin/matlab/JavaMatlab.html`)
17. Xsens Ltd.: Motion Technologies. (Company Website: `www.xsens.com`)
18. Polastre, J., Szewczyk, R., Culler, D.: Telos: Enabling ultra-low power wireless research. In: Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS). (2005)
19. Crossbow Technology Inc.: Solutions for Wireless Sensor Networks. (Company Website: `www.xbow.com`)
20. Szewczyk, R., Polastre, J., Mainwaring, A.M., Culler, D.E.: Lessons from a sensor network expedition. In Karl, H., Willig, A., Wolisz, A., eds.: EWSN. Volume 2920 of Lecture Notes in Computer Science, Springer (2004) 307–322
21. Moteiv Corporation: Accelerating Sensor Networking. (Company Website: `www.moteiv.com`)
22. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., Pister, K.S.J.: System architecture directions for networked sensors. In: Architectural Support for Programming Languages and Operating Systems. (2000) 93–104
23. TinyOS: An Open-Source Operating System Designed for Wireless Embedded Sensor Networks. (Project Website: `www.tinyos.net`)
24. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems (2003)
25. Wildlife Computers: Innovative Tags for Innovative Research. (Company Website: `www.wildlifecomputers.com`)
26. Ubisense Ltd.: The Ubisense UWB Localization System. (Company Website: `www.ubisense.net`)
27. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: An acquisitional query processing system for sensor networks. ACM Transactions on Database Systems **30**(1) (2005) 122Ũ173
28. Intel Research Laboratories Berkeley: Intel Lab Data. (Project Website: `berkeley.intel-research.net/labdata/`)
29. Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J., Hong, W.: Model-driven data acquisition in sensor networks. In: Proceedings of the 30th Very Large Data Base Conference (VLDB), Toronto, Canada (2004)
30. Santini, S., Römer, K.: An adaptive strategy for quality-based data reduction in wireless sensor networks. In: Proceedings of the 3rd Intl. Conf. on Networked Sensing Systems (INSS 2006), Chicago, IL, USA (2006)