Dandelion: A Framework for Transparently Programming Phone-Centered Wireless Body Sensor Applications for Health

Felix Xiaozhu Lin[1], Ahmad Rahmati[2], and Lin Zhong[1,2]
[1]Dept. of Computer Science and [2]Dept. of Electrical & Computer Engineering, Rice University
Houston, TX, USA
{xzl, rahmati, lzhong}@rice.edu

Abstract

Many innovative mobile health applications can be enabled by augmenting wireless body sensors to mobile phones, e.g. monitoring personal fitness with on-body accelerometer and EKG sensors. However, it is difficult for the majority of smartphone developers to program wireless body sensors directly; current sensor nodes require developers to master node-level programming, implement the communication between the smartphone and sensors, and even learn new languages. The large gap between existing programming styles for smartphones and sensors prevents body sensors from being widely adopted by smartphone applications, despite the burgeoning Apple App Store and Android Market.

To bridge this programming gap, we present *Dandelion*[1], a novel framework for developing wireless body sensor applications on smartphones. Dandelion provides three major benefits: 1) platform-agnostic programming abstraction for in-sensor data processing, called *senselet*, 2) transparent integration of senselets and the smartphone code, and 3) platform-independent development and distribution of senselets.

We provide an implementation of Dandelion on the Maemo Linux smartphone platform and the Rice Orbit body sensor platform. We evaluate Dandelion by implementing real-world applications, and show that Dandelion effectively eliminates the programming gap and significantly reduces the development efforts. We further show that Dandelion incurs a very small overhead; in total less than 5% of the memory capacity and less than 3% of the processor time of a typical ultra low power sensor.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems – *distributed applications*

General Terms

Design, Experimentation, Performance

1. Introduction

Six in ten people (60%) of world's population have mobile phones [1]. A significant and growing percentage of mobile phones are smartphones that enjoy mobile Internet, powerful processor, and massive storage. Because mobile users usually keep their phones within the arm's reach [2], many have envisioned that smartphone-like devices serve as the personal hub for body sensors, bridging the sensors with human users and the Internet [3]. Furthermore, wireless body sensors naturally extend the "sense" of a smartphone and open doors to new applications, especially in healthcare. For example, wireless body sensors can measure physical activity, collect physiological data, and even infer social context. Such information provides key insights into the health and lifestyle of the user and can potentially help the user make day-to-day *in situ* decisions that have an impact on their health. Numerous research prototypes have been reported in literature, and there has been a few successful commercial health applications in recent years, e.g. the Nike+iPod Sport Kit [4] and the WIN human recorder [5].

Even so, only a few of the millions of third-party smartphone applications employ body sensors, despite the commercial availability of Bluetooth body sensors. We contribute the lack of body-sensor applications for smartphones to the disparity between programming a smartphone and programming wireless sensors, which requires smartphone developers to directly program a sensor node with a new programming language.

Toward addressing this challenge, we seek to enable smartphone developers to easily implement data processing that executes in wireless sensors, without learning the particular programming style coupled with the target sensor platform. Our solution, called *Dandelion*, is a framework combining both programming support and execution support for in-sensor data processing. The key features of Dandelion are:

- A platform-agnostic, smartphone-style programming abstraction called *senselet*;
- A mechanism for transparently integrating senselets with the smartphone application; and
- A platform-independent mechanism for distributing and deploying senselet executables.

[1] A plant best known for its tiny, airborne seeds, which remind the authors of wireless body sensors

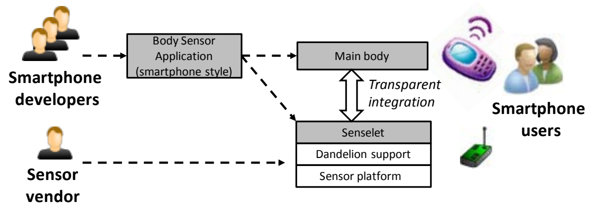


Figure 1. The ecosystem for smartphone health applications promoted by Dandelion

With these features, Dandelion allows smartphone developers to program in-sensor data processing using the smartphone programming style in senselets. Dandelion provides the transparency in two key aspects: *programming style* transparency, which makes developing senselets very close to developing traditional smartphone applications; *target platform* transparency, which makes the development, distribution, and deployment of senselets independent of any particular sensor platform. Throughout this paper, we use the term *sensor platform*, or simply *platform*, to refer to the specific combination of hardware and software of a sensor node, unless specified otherwise.

We prototype Dandelion with the Maemo Linux smartphone platform, used by Nokia N900 and N810, and the Rice Orbit body sensor platform. We experimentally evaluate our prototype with real-world applications including Fall-detector, Pedometer, and EKG monitoring. Our evaluations show that the Dandelion framework significantly reduces the application developers' burden comparing with existing methods of directly programming sensors. Furthermore, Dandelion incurs an overhead of less than 5% of the memory capacity and less than 3% of the processor time of a typical low-power body sensor.

Dandelion supports a healthy ecosystem that promotes the interests of all three parties involved in body sensor applications: the sensor vendors, the smartphone developers, and the end users, as illustrated in Figure 1. 1) A sensor vendor provides lightweight Dandelion support for its sensor. Because its sensor can be easily programmed by any developer, the vendor will see more smartphone applications for its sensors. 2) A smartphone developer only needs to program senselets as part of the smartphone application without learning the specifics of any particular sensor platform or programming style. Because such applications can transparently work with any sensor that has Dandelion support, the developer benefits by reaching a large number of users who may choose different sensors. 3) Finally, smartphone users can benefit from a more flexible choice of sensors for their intended applications, as long as the sensors have the Dandelion support. Consequently, users will also benefit from a wider range of emerging applications and sensors.

The rest of the paper is organized as follows. We analyze the challenges towards the development of smartphone body sensor applications and motivate Dandelion design in Section 2. We then present the programming and execution support of Dandelion in Sections 3 and 4, respectively. We present our prototype based on a Nokia N900 smartphone and Rice Orbit sensors in Section 5, and evaluate Dandelion in Section 6. We discuss related work in Section 7 and finally conclude in Section 8.

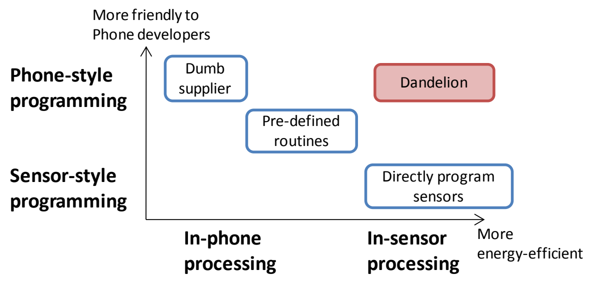


Figure 2. The tension in developing sensor data processing for smartphone body sensor applications

2. Motivation

Among millions of smartphone applications, only a few employ body sensors. In this section we first highlight the challenges faced by smartphone developers wanting to use body sensors, and then motivate the design of Dandelion by showing the limitations of existing solutions.

2.1 The Gap between Two Programming Styles

Figure 2 shows the tension between system energy efficiency and ease of body sensor application development. At one extreme, (ease of development), the developer can treat the sensors as dumb data suppliers and processes the raw data on the smartphone ("Dumb supplier" in the figure). Figure 3 uses a skeleton code to illustrate the smartphone style for processing sensor data, which uses a clear interface in the object-oriented fashion. While very friendly to smartphone developers, this paradigm is extremely energy inefficient because moving excessive raw data over the wireless link consumes a lot of energy, and consequently reduces the battery lifetime of the phone and sensors.

To improve the efficiency, many research prototypes, e.g. [6] [7], implement a fixed set of sensor routines to support a pre-defined set of data processing in the sensor ("Pre-defined routines" in Figure 2). This is similar to the Application Profiles provided by Bluetooth. While this approach improves energy efficiency for some applications, its lack of sensor programmability inevitably restricts developers to create applications that require novel, application-specific data processing.

At the other extreme, the application developers can carefully partition the data processing between the phone and the sensors, and then program both *directly*. While this paradigm can be very efficient, it usually requires the developer to master sensor *node-level* programming, with low-level abstractions that are tightly coupled with various underlying platforms [8-11], and probably involves a different programming language. For example, TinyOS requires devel-

```
class MySensorListener : SensorListener {
public:
  // called when the processing starts
  OnCreate() { ... };
  // called when the processing stops
  OnDestroy() { ... };
  // called when new sensor data is acquired
  OnNewData(sensor_id, data) { ... };
private:
  // private states as variables
}
```

Figure 3. Skeleton code for data processing on smartphone. This style for sensor data processing shown here is widely adopted by smartphone programming frameworks, including Android, iPhone OS, Symbian S60, and Maemo

opers using nesC language [12] to write sensor code as a set of software components. In addition, TinyOS requires developers to define events and commands as the interface for each component, and properly wire interfaces together. Although very effective at programming nodes in wireless sensor networks, such programming requirements are foreign to many smartphone developers and are significantly different from existing smartphone programming styles. Macro-programming models at the *network-level* [13-15] target relatively large-scale sensor networks. The introduction of new language constructs such as new statements, predicates, rules, or task graphs, makes them even more difficult for smartphone programmers to manage. In short, the gap between the smartphone programming style and that of various sensor platforms constitutes a practical barrier for the majority of smartphone developers to leverage body sensors and create applications.

2.2 Overview of Dandelion

Our goal is to achieve both high efficiency and ease of development by making sensor programming transparent to the smartphone developers (“Dandelion” in Figure 2). By “transparent”, we mean that smartphone developers do not have to deal with the native programming abstraction or hardware/software specifications of the sensor. Our solution toward this goal is Dandelion, a framework that allows smartphone application developers to “program” body sensors by simply writing the code as a smartphone software module. Dandelion achieves such design goal with *senselet*, a smartphone-style, platform-agnostic programming abstraction for in-sensor data processing.

Dandelion supports this abstraction from both the *programming* aspect (Section 3) and the *execution* aspect (Section 4). For programming support, Dandelion defines the degree of transparency with three key design decisions: a compact set of *platform services* for senselet to access platform resource, the *Remote Method Invocation* (RMI) mechanism for integrating senselets with the application code running on the smartphone (called *main body* in this work), and the *two-phase compilation* technique for generating senselet executable.

```
class MySenselet : public SenseletBase {
public:
  // App-specific initialization
  void OnCreate() {...};
  // App-specific finalization
  void OnDestroy() {...};

  // Receive and process new sensor data
  void OnData(data) {...};
private:
  // All senselet states are private variables
}

// ... in the main body code ...
LoadSenseletInstance(MySenselet);
```

Figure 4. The skeleton code of a Senselet class

For execution support, Dandelion manages and executes senselets with its distributed middle layer or *runtime system*. The runtime system consists of a smartphone runtime component and one or more sensor runtime components. Runtime components communicate with messages. The smartphone runtime acts as the coordinator to command all sensor runtimes, and a sensor runtime fulfills such commands and executes the senselet code.

3. Programming Support

Toward making sensor programming transparent, Dandelion targets *practical* transparency. A *complete* transparency is ideal to the developer: the development and execution of a senselet will be indistinguishable from a smartphone software module. However, such complete transparency is very expensive, if not impossible, given the huge discrepancy between the hardware and software environments on the smartphone and on body sensors. For example, implementing coherent shared memory across the smartphone and sensors incurs prohibitive overhead.

Dandelion defines practical transparency for senselet with four design decisions: 1) hide platform-dependent programming styles inside the template (Section 3.1), 2) choose a compact set of platform services as the unified interface to platform resources (Section 3.2), 3) use the RMI mechanism to transparently integrate senselets into the smartphone application (Section 3.3), and 4) compile and distribute senselets as platform-independent intermediate representations (IRs) and translate the IRs into platform-specific binaries during application installation (Section 3.4). We will discuss the four decisions in details below.

3.1 Template for Senselet

Senselet is the core of Dandelion that abstracts in-sensor data processing. As shown in the object-oriented skeleton code in Figure 4, a senselet is a subclass that inherits the virtual base SenseletBase. SenseletBase declares a concise interface as a set of virtual methods for concrete senselets. For example, the senselet class overrides OnCreate() to initialize its states, and overrides OnData() to receive new sensor data for processing. Compared to various sensor native programming abstractions mentioned in Section 2.1, the

```
SenseletBase::_EventLoop (event_t *event){
  switch(event->type){
  case EVENT_INIT:
    //...platform-dependent initialization
    OnCreate();
    return;
  case EVENT_DATA:
    //...decode data from the event
    OnData(data);
    return;
  case EVENT_RMI:
    //...call corresponding RMI stubs
    return;
  }
}
//Assume MySenselet points to Senselet instance
// Start an OS task with the event loop
start_task(MySenselet->_EventLoop);
```

Figure 6. Pseudo code of a simplified SenseletBase, as a template to wrap an event loop skeleton for Senselet class

senselet abstraction has three distinct features: 1) a just-fit interface for data processing, 2) smartphone style programming (very close to the skeleton code shown in Figure 3), and 3) platform-independence.

SenseletBase contains the *template* code written with sensor native programming abstraction, while leaving the implementation of the actual data processing to senselets. Essentially, a SenseletBase class plays two important roles. First, it acts as an adaptor between the senselet abstraction and the sensor native programming abstraction. Second, it hides low-level platform configurations, e.g. hardware parameters from the developer.

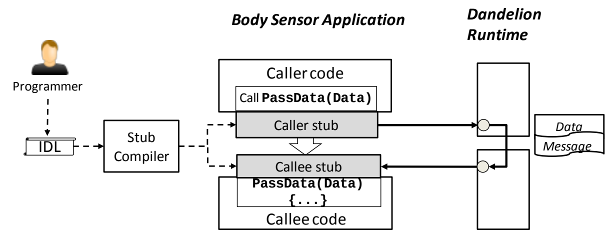
Since the implementation of a SenseletBase is platform-specific, it is the sensor vendor's responsibility to provide SenseletBase as a basic library. When a Dandelion application is being installed on a smartphone, the developer-provided senselet code is linked with vendor-provided SenseletBase to generate the final senselet executable, as will be further addressed in Section 3.3.

Example: We use a simple example in Figure 5 to illustrate how SenseletBase works as a template. Suppose that the sensor platform requires any of its software modules to be coded as an event loop. SenseletBase populates such an event loop in the function _EventLoop(), with dispatching events to the proper methods implemented by a concrete Senselet class (e.g. OnStart and OnData).

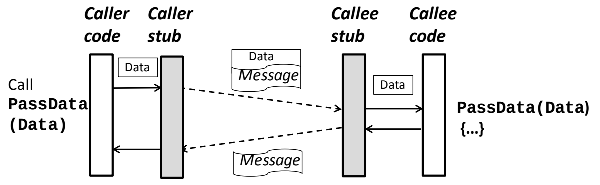
3.2 Platform Services

Dandelion abstracts platform resource as a set of *platform services*, which resemble system calls in traditional OS. In designing platform services, we seek to strike a balance between their portability and the flexibility of programming; more platform services allow the developer to program senselet in a more flexible way, but they may be supported by fewer platforms.

By examining data processing in a wide range of body sensor applications, both in the market and reported in literature, we identify three core platform services for senselets, as follows.



(a)



(b)

Figure 5. The internals of an example RMI PassData(). (a) The stub compiler generates both caller and callee stubs that translate the RMI into messages. Both main body and senselet can be caller or callee. (b) During the RMI, the caller transfers the data and its control flow to the callee

Acquire sensor readings. Get raw sensor readings, using either *pull* mode, where senselet actively polls for readings:

```
data = PollSensorData(sensor_id);
```

or *push* mode, where the senselet requests runtime to periodically deliver readings, with:

```
RegisterSensorData(sensor_id, rate);
```

Sensor runtime delivers new reading to the senselet by invoking its `OnData()` method.

Timer. A senselet may need to process data with timing, periodically or aperiodically. A senselet can register a timer with:

```
timer_id = RegisterTimer (interval)
```

Consequently, sensor runtime will invoke its `OnTimer()` method at the desired interval. The senselet can also use `UnRegisterTimer(timer_id)` to cancel a registered timer.

Memory management. A senselet can use `Malloc()` to request dynamic memory for storing intermediate processing results, and use `Free()` to release the memory after use.

Finally, we note that new services can be added as they become critical to data processing and widely supported by sensor platforms, e.g. storage.

3.3 Remote Method Invocations

A senselet executes on a body sensor, while the main application body still runs on the smartphone. However, to logically work as an integral application, they need 1) transfer data to each other, e.g. a senselet passes preliminary results to the main body for further processing, and 2) transfer control flow to each other, e.g. the main body commands to change the sampling rate used by a senselet and waits for a confirmation from it. Rather than require developers to hand-code integration with messages, Dandelion employs *remote method invocation* as the solution to address both requirements.

RMI is a cross-platform communication mechanism widely used in distributed systems such as Network File System. From the perspective of a developer, an RMI that happens between a senselet and the main body in Dandelion, appears to be very similar to a *local* call. All a developer needs to do is to specify the name and types of parameters used by the remote method, using Interface Description Language (IDL) [16]. All RMIs in Dandelion are synchronous, i.e. the caller returns after the remote method finishes execution.

We provide a *stub compiler* that automatically generates the implementation of two stub procedures for a RMI, based on its IDL description. One stub is used by the caller to issue the RMI, with the data objects to transfer as method parameters. When the caller stub is invoked, it creates a message and marshals method parameters into the message payload. The other stub procedure is used by the callee to un-marshal the parameters from the received message and actually calls the remote method. Figure 6(a) illustrates the roles of the programmer, stub compiler, and runtime in implementing a RMI.

We choose RMI for integration because of two reasons. First, it is a widely used way for separated execution contexts to interact with each other. Second, IDL is already an essential part in smartphone development, e.g. AIDL in Android [17], XML for dbus in Maemo [18]. Recent proposal to offload execution from mobile devices also leverage IDL-based methods to serialize passed parameters [19].

Shared memory is an alternative solution for data transfer: smartphone and sensor runtimes collaboratively implement a memory coherence protocol by exchanging messages. With this scheme, senselets and the main body can access shared variables. While it may be desirable to support shared memory for smartphone developers who may be unfamiliar with distributed programming, its overhead is significant for body sensors, both in terms of implementation complexity and communication (i.e. frequently sending small memory updates over wireless link).

Example: We use a simple but general example to show how RMI transfers data and control flow. In this example, the senselet calls a remote method defined in the main body, named `PassData`, to transfer processed data. The programmer describes the following method interface using XML-style IDL [18]:

```
<method name=”PassData”>
    <arg type=”ai” name=”Data” direction=”in” />
</method>
```

The IDL description above indicates that the method `PassData` carries one parameter `Data`, which is an integer array

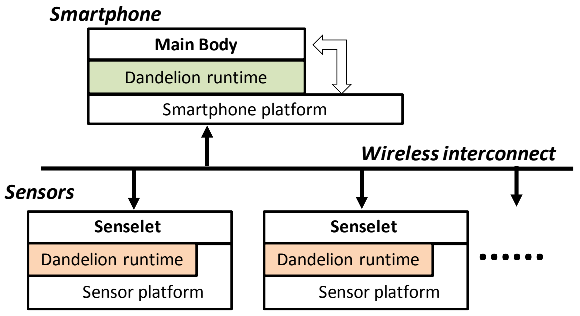


Figure 7. The overview of Dandelion runtime architecture

(‘`ai`’). At compile time, our stub compiler generates the caller stub for the senselet:

```
void PassData (int *Data, int actual_len) {...};
```

The senselet code simply calls `PassData()`provided by the caller stub to send the integer array, and the main body accordingly implements the method `PassData()`to receive the array. It is worth noting that during the RMI, the senselet code in fact transfers its control flow to the main body (as shown in Figure 6(b)). Therefore, upon returning from `PassData()`, the senselet is ensured that the execution of remote method in the main body is finished. This guarantee is especially important when the main body uses RMI to configure the senselet, e.g. change the data sampling rate.

3.4 Senselet Executable Generation

The challenge of generating senselet executable comes from our design goal of platform transparency: platforms may have various Instruction Set Architectures (ISAs) that developers may not know about. One possible solution to bridge the ISA gap is Virtual Machine (VM). With the same VM installed on all target platforms, a senselet can be compiled into VM byte code and then interpreted by a VM during execution. Although light-weight VMs [20] have been developed for low-power processors to perform simple control tasks, the overhead of VM interpretation is still high for intensive data processing (10 times or more compared to native binary [21]).

Therefore, Dandelion adopts the *two-phase compilation* technique that has been used for other heterogeneous distributed systems such as [22]. A developer compiles a senselet into intermediate representation (IR) before distribution (first phase). When a smartphone user installs the distribution (second phase) on her phone, the corresponding cross-compiler for her sensor translates the IR into sensor native binary. This technique relieves the application developer from worrying about various sensor ISAs while achieving the high performance of native execution.

4. Execution Support

The architecture of a running Dandelion system is shown in Figure 7. Dandelion employs a distributed middle layer or *runtime* to manage the execution and communication of senselets. It also provides *resource exception* to support application-specific handling of resource depletion during senselet execution.

4.1 Dandelion Runtime System

The Dandelion runtime, as the core of execution support, consists of multiple component runtimes: one smartphone runtime and one or more sensor runtimes. Component runtimes communicate with each other through messages. They employ a common message format to serve both RMIs in application code and runtime management functions. While messaging is the building block of Dandelion communication, it is only visible inside the runtime system and totally transparent to the application.

4.1.1 Smartphone Runtime

As the coordinator of the runtime system, the smartphone runtime serves three key management functions. First, when a body sensor application starts, the smartphone runtime discovers sensors by querying sensor runtimes. Second, the smartphone runtime manages the network connections between the smartphone and sensors according to the application requirement. It does so with existing networking capability, e.g. Bluetooth Serial Port Profile. Finally, the smartphone runtime sends senselet native binaries to the corresponding sensor runtimes for execution and stops senselet execution under the application's request. Again, the communication between the smartphone runtime and the sensor runtimes is based on messages described above.

4.1.2 Sensor Runtime

To minimize the complexity and execution overhead of the sensor software stack, we opt a minimalist design for sensor runtime. A sensor runtime provides two core functions: 1) it implements the very small set of platform services for senselets, as described in Section 3.2 and 2) it fulfills management commands from the smartphone runtime. For function 2, to further reduce the complexity of sensor runtime, we carefully leave most functions to smartphone runtime. For instance, to load a new senselet for execution, we perform the dynamic linking of the executable on the smartphone instead of within the sensor runtime [21], so that the sensor runtime only needs to receive and copy the linked executable into sensor memory.

With this minimalist design, the sensor runtime is very lightweight in terms of memory and processor usage. Furthermore, the sensor runtime only requires a minimal set of primitives, namely interrupt handling, digitalized sensor output, and the configurable timer, from the underlying platform. As a result, the Dandelion sensor runtime can be built on top of various embedded OSes, e.g. uC/OSII, Mantis, TinyOS, SOS, Contiki, and with diverse sensor hardware, e.g. MSP430, ARM7, AVR. We will evaluate the sensor runtime design in Section 6.

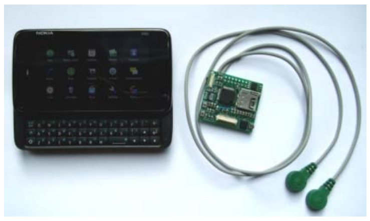


Figure 8. The Dandelion prototype hardware, based on Nokia N900 smartphone (left) and Rice Orbit body sensor (right)

4.2 Senselet Resource Exception

Due to the highly limited computing resources on body sensors, a senselet is likely to run out of resources. Unfortunately, it can be hard for developers to match senselet resource demand with available platform resources. On one hand, developers may have limited knowledge about target platforms; on the other hand, senselet resource demands may vary significantly during its execution (e.g. changing sampling rate with human activity).

Dandelion supports *resource exception* to let developers gracefully deal with such resource mismatch during execution, in an application-specific manner. We define two types of resource exceptions for senselet:

- *TimeException* is raised when running out of processor time: e.g. senselet fails to process the sensor data faster than the data is supplied;
- *MemException* is raised when running out of memory: e.g. the stack is going to overflow.

The sensor runtime monitors platform resource utilization in a platform-specific way, e.g. read processor utilization from the scheduler, and raises exceptions by invoking corresponding exception handlers in senselet. Resource exception provides enough flexibility for application code to adapt to constrained sensor resource. For instance, when TimeException is raised, senselet code may lower its sampling rate and notify the main body with a RMI.

5. Prototype Realization

We have built a Dandelion prototype using Nokia N900, a smartphone based on Maemo Linux, and Rice Orbit, a multi-purpose sensor platform with a TI MSP430 microcontroller and a Bluetooth interface. Rice Orbit also has a tri-axis accelerometer that can be utilized for human activity monitoring, and an amplified analog input that can be used to sample Electrocardiogram (EKG) signals. Rice Orbit works with a variety of embedded OSes that support TI MSP430, including µC/OS-II, SOS, Contiki, and TinyOS.

Smartphone-side software

We implement the smartphone runtime using around 1000 lines of C++ code and link it as a static library with the main body. The smartphone runtime only requires network capability from the smartphone platform, therefore it is highly portable and can therefore be realized with all current

```
class SenseletFall : public SenseletBase {
public:
  SenseletFall () {_avg_energy = 0;};

  void OnCreate() {RegisterSensorData(ACCEL, 50);};

  void OnData(uint8_t *readings, uint16_t len) {
    uint16_t energy = readings[0]*readings[0] + \
                      readings[1]*readings[1] + \
                      readings[2]*readings[2];
    //do a simple low-pass filtering
    _avg_energy = _avg_energy / 2 + energy / 2;

    // detect fall accident with the filtered energy
    if (_avg_energy > THRESHOLD) {
      theMainBody.FallAlert();  //RMI
    }
  }

  void OnDestroy() {UnRegisterSensorData(ACCEL);};

private:
  uint16_t _avg_energy;
};
```

Figure 9. **An implementation of FallDetector with senselet**

smartphone platforms we are aware of, including Maemo, iPhone, Android, and S60.

We have built a stub compiler that takes IDL description to generate RMI stub source code for both the main body and senselets. We implement such stub compiler with around 500 lines of Python code, by referring to dbus implementation [23].

We have validated the feasibility of two-phase compilation by using LLVM-msp430, a high-efficient compiler infrastructure with an experimental MSP430 backend, to successfully build simple senselets with two phases. However, since the MSP430 backend is under heavy development, in the evaluation we still choose MSPGCC, the widely-used MSP430 port of gcc, as the C/C++ compiler kit to produce senselet binaries. As LLVM-msp430 matures, we expect it to produce production-quality MSP430 binary with two-phase compilation.

Sensor-side software

We implement the Dandelion sensor runtime on top of µC/OS-II [24], a popular, lean OS for resource-constrained embedded systems. Since µC/OS-II provides no direct support for any of three platform service, we only rely on it for multitasking and implement all platform services inside sensor runtime. We use a µC/OS-II task to implement sensor runtime management functions, and use another task to execute senselet code (with a fixed 64 byte stack).

The sensor runtime is highly portable, thanks to its minimalist design. Its two major components, platform services and management functions can be similarly realized on top of most embedded OSes. In many cases, the platform services are just thin wrappers over existing OS support. It is worth noting that protothreads [11] can be used to implement blocking in synchronous RMI with a negligible overhead of 2 bytes, which is useful for purely event-driven OSes that lack direct support of blocking. What's more, even without an underlying OS, it is possible to build sensor runtime on bare-metal with the following four hardware primitives: 1) hardware interrupts, 2) access to digitalized sensor data, e.g. through a built-in ADC or a digital interface with the sensing apparatus, 3) a configurable timer, and 4) a bi-directional, interrupt-enabled communication port that enables network capability. Such hardware primitives are available on most 16-bit and even some 8-bit microcontrollers.

6. Evaluation

To evaluate if Dandelion achieves its design goal, we develop in-sensor data processing from three real-world smartphone health applications and in three alternative styles: 1) the bare-bone style, without embedded OS or Dandelion; 2) the embedded-OS style, implemented with an event loop; 3) the Dandelion style, implemented as senselets. We compare the developers' burdens in coding in these three styles by examining the resulted source code. To understand the cost in supporting transparency, we also experimentally quantify the memory and execution overhead of senselet and Dandelion runtime.

6.1 Benchmark Health Applications

We use the following three real-world applications in the benchmark:

FallDectector uses a wearable accelerometer to detect fall accidents occurred to the user and raises an alert to the main body accordingly. The main body reacts by sending out an SMS to registered phone numbers..

The *EKG* application [25] monitors the user's heart rate by calculating the average RR interval in the real-time EKG trace. The main body can retrieve the calculated heart rate, or raw EKG traces for further analysis.

Pedometer [26] uses a shoe-mounted accelerometer to count user steps and recognize walking distance, and further calculates walking speed and consumed calorie. The main body regulates the thresholds for steps, speed, or consumed calorie, all calculated in the senselet. The senselet in turn notifies the main body when any quantity exceeds its threshold. The main body can also poll these quantities.

6.2 Source Code Examination

We use *FallDetector* as the simplest example to compare developer's burdens in three programming styles. The data processing is simple: measure the acceleration energy (calculated as the magnitude of X, Y, Z readings) and apply a simple low-pass filter to the energy trace. If the filtered energy exceeds a pre-defined threshold, an alert is raised to the main body.

The FallDetector senselet is shown in Figure 9, which consists of only 17 lines of C++ code. The OnCreate() method requests periodic accelerometer readings at 50 Hz. Later on, OnData()is invoked by sensor runtime when new reading is acquired. OnData() detects fall accident and raises the alert

Table 1. The source lines of code in developing in-sensor data processing, with different programming styles

Style	FallDetector	EKG	Pedometer
Dandelion	17	96	158
Embedded OS	72	146	248
Bare-bone	84	194	296

Table 2. Memory breakdown of sensor runtime, in byte

Module	RAM	ROM
Sensor reading service	12	144
Timer service	36	250
Memory management service	4	248
Message communication	64	286
Management functions	64	818
Total	180	1746

Table 3. Memory overhead of senselet executables, in byte and the percentage of the whole sensor memory. Assume whole memory of 10KB RAM and 48KB ROM

Code part	FallDetector		EKG		Pedometer	
	RAM	ROM	RAM	ROM	RAM	ROM
Template	64	324	64	324	64	324
RMI stubs	0	54	0	56	0	210
Total	64	378	64	380	64	534
% of whole	0.6%	0.8%	0.7%	0.8%	0.7%	1.1%

by invoking the remote method FallAlert() of the main body. Senselet abstraction relieves the developer from dealing with hardware and communication details.

For comparison, we also write FallDetector in the form of a main event loop, which is used for tasks with many embedded OSes, e.g. SOS, Contiki, and µC/OS-II. In developing the code, we favorably assume that the OS provides all peripheral drivers, event queues, and supports blocking, which may not be true for all. Even with this favorable assumption, the same data processing takes three times as many lines of code. The bare-metal implementation is even more verbose. While it also implements a main loop driven by several hardware interrupts to perform processing and communication, it has to deal all low-level, hardware-related operations, e.g. manually triggering the power state transitions.

The same observations also apply to other two benchmark applications (source statistics in Table 1). As reflected in the numbers of source lines, without Dandelion support, complex data processing and communication requires more programming efforts; what is even worse, the resulting code is non-portable and error-prone.

By further comparing the source code, we can see Dandelion saves considerable development efforts in three aspects. First, senselet only exposes to developers a very concise interface necessary for in-sensor data processing. The other two direct sensor programming styles put extra burden on developers: populate a main loop woven with the data processing and drive the loop with (sometimes low-level) events.

Second, the use of RMI facilitates integration and relieves the developer from dealing with smartphone-sensor communication. Compared to issuing a RMI with a single line in senselet, the other two styles need to spend 30-50 lines of code to manually construct a message and send through the wireless link, when accounting for link reliability. Additionally, the bare-bone code has to manage the power state of Bluetooth interface.

Third, platform independence makes senselet code portable. For example, to periodically acquire sensor readings; embedded OS style requires using timer event to drive the acquisition, while the bare-bone style additionally requires manually configure ADC. In contrast, Dandelion sensor reading service covers these differences.

6.3 Overhead Measurements

We measure the overhead of Dandelion framework in terms of both memory consumption and processor cycle consumption. In the measurement, we compiled all code using MSPGCC version 3.2.3, with optimization level -O2. We are aware of that the overhead is affected by the choice of the underlying OS: complete support for runtime functionalities from OS can largely simplify sensor runtime implementation and therefore reduces its overhead. In the evaluation, we estimate the upper limit of runtime overhead by using µC/OS-II merely for multitasking, and implement most sensor runtime functionalities from scratch, instead of choosing many other embedded OSes with existing support for peripheral drivers, timer, or communication.

6.3.1 Memory Overhead

We break down the memory overhead of Dandelion into two parts: 1) that of sensor runtime, and 2) that of senselet executable, including template and RMI stubs, as shown in Table 2 and Table 3.

The runtime memory overhead, due to the minimalist design, is a small constant: 180 bytes of RAM and 1746 bytes of ROM. For the second part, in a single senselet executable, template takes a small, fixed amount memory. Additionally, each RMI stub only uses 50-100 bytes ROM. For a typical sensor controller, the TI MSP430F161 (10KB RAM, 48KB ROM) that is widely used in various sensors such as Rice Orbit and Shimmer[27], the fixed overhead is less than 5% of the whole memory.

The measurements have two implications. First, the small overall memory overhead allows Dandelion be implemented with many resource-constrained sensors. Second, the compact template and RMI stubs incur small energy overhead when a senselet executable is transmitted over wireless link.

6.3.2 Execution Overhead

To gain insight into the execution overhead of a senselet, we first measure execution overhead in common operations of sensor runtime. Note that we do not include cycles spent on sending and receiving message bytes into message RX/TX overhead. Further, the overhead of an RMI varies with the number and types of its parameters, and 300 cycles are typical for passing an integer array. Results in Table 4 show that the sensor runtime incurs small execution overhead; with a typical processor clock rate of 4MHz, even the most costly operation, message RX, takes less than 250 us.

To estimate the execution overhead in long-run, we further measure the performance of the most common execution path in senselets. Since in all three senselets data processing is driven by periodic sampling, we define such path as data processing performed in each sampling period (without RMI involved), e.g. EKG senselet reads in new data and computes to determine if it is in R wave. The results in Table 5 show that, even as the data processing becomes intensive, Dandelion execution overhead remains small (around 250 cycles). Even with the most challenging EKG benchmark, where data processing consumes around 3900 cycles (~50% of processor time) in every 2 ms sampling period, the execution overhead only takes ~3% processor time. We have observed that the execution overhead comes from timer notification, sensor reading, and the template code.

Table 4. Execution overhead in common sensor runtime operations, in processor cycles

Operation	#Cycles	Operations	# Cycles
Timer notification	53	Sensor reading	130-168
RMI	50-300	Load senselet	716
Message RX	924	Message TX	720
Malloc	239	Free	120

Table 5. Execution overhead in the most common data processing path, in processor cycles and the percentage of processor time in each sampling period. FallDetector and Pedometer acquire sensor data at 50Hz, while EKG does at 512Hz. Assume a clock rate of 4MHz

Execution overhead	FallDetector	EKG	Pedometer
# Cycles	251	221	251
% per sampling period	0.3%	3%	0.3%

7. Related work

Dandelion allows a smartphone developer to use body sensors in her program without directly programming the sensors or dealing with the sensor hardware/software. It is related to three seemingly disparate groups of work.

The first group provides programming frameworks to body sensor networks, including MobiCare [28], CodeBlue [29], and SPINE [6]. Some of them facilitate application development by providing pre-defined functions in sensors. For example, SPINE provides a portable library for in-sensor signal processing. While they are effective for particular applications that use such pre-defined functions, compared to Dandelion, they have very limited sensor programmability, as discussed in Section 2.1.

The second group of related work addresses the programming models of wireless sensor networks (WSN) in general. Solutions such as nesC [12] and protothreads [11] provide node-level programming abstractions. Macro-programming [13-15] enables developing WSN applications by describing the network-level behavior. While successful for complicated WSN applications, such programming styles significantly differ from those used in smartphone application development, making it is difficult for smartphone developers to adopt them. In contrast, Dandelion leverages the simplicity of smartphone-centered body sensor networks and focuses on supporting in-sensor data processing tasks. With this trade-off, Dandelion is able to provide transparency in programming style.

Finally, the third group of related work employs heterogeneous distributed systems and support different levels of programming transparency. In most cases, e.g. [30-32], the programmer has to program each platform in the system directly without any transparency. Some systems support programming transparency with an unified OS abstraction or distributed runtime system, mostly based on a virtual-machine approach [22] to hide ISA variances. This approach, however, proves to be inefficient on resource-constrained sensors [20, 21]. In contrast, Dandelion achieves transparency by limiting the senselet functions to data processing and by introducing an extra compilation phase to produce sensor native binaries during application installation.

8. Conclusion

We present Dandelion, a novel programming framework for phone-centered wireless body sensor applications. Dandelion allows developers to easily write data processing code to be executed on sensors, in a programming style similar to traditional smartphone development. With the minimalist design of the runtime system, Dandelion incurs very small overhead and therefore can be easily ported to various resource-constrained sensor platforms. We believe that by enabling smartphone developers to easily write body sensor code, Dandelion supports a healthy ecosystem that promotes the interests of all involved parties: sensor vendors, smartphone developers, and users.

Acknowledgements

The work was supported in part by NSF Grants #0713249, #0751173, #0803556, and #0923479, and by the Texas Instruments Leadership University program. The authors were also grateful to the anonymous reviewers whose constructive comments help improve the final version.

References

[1] R. C. Hodgin, "60% of world's population now has cell phone, highest ever," in *TG Daily*, 2009.

[2] S. Patel, J. Kientz, G. Hayes, S. Bhat, and G. Abowd, "Farther than you may think: An empirical investigation of the proximity of users to their mobile phones," in *Proc. Ubicomp*, 2006, pp. 123-140.

[3] L. Zhong, M. Sinclair, and R. Bittner, "A phone-centered body sensor network platform: cost, energy efficiency & user interface," in *Proc. Int. Wrkshp. Wearable and Implantable Body Sensor Networks*, 2006.

[4] Apple, "Nike + iPod Sport Kit," 2010.

[5] S. Kato, "Wearable Health Monitoring Sensor Debuts in Japanese Market," 2010.

[6] P. Kuryloski, A. Giani, R. Giannantonio, K. Gilani, R. Gravina, V.-P. Seppa, E. Seto, V. Shia, C. Wang, P. Yan, A. Y. Yang, J. Hyttinen, S. Sastry, S. Wicker, and R. Bajcsy, "DexterNet: An Open Platform for Heterogeneous Body Sensor Networks and its Applications," in *Proceedings of the 2009 Sixth International Workshop on Wearable and Implantable Body Sensor Networks*: IEEE Computer Society, 2009.

[7] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, "The Tenet architecture for tiered sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems* Boulder, Colorado, USA: ACM, 2006.

[8] D. Gay, P. Levis, and D. Culler, "Software design patterns for TinyOS," *ACM Trans. Embed. Comput. Syst.*, vol. 6, p. 22, 2007.

[9] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services* Seattle, Washington: ACM, 2005.

[10] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh, "Resource aware programming in the Pixie OS," in *Proc. ACM Conf. Embedded Networked Sensor Systems (SenSys)* Raleigh, NC: ACM, 2008, pp. 211-224.

[11] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems* Boulder, Colorado, USA: ACM, 2006.

[12] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler, "The *nesC* language: A holistic approach to networked embedded systems," *ACM SIGPLAN Notices*, vol. 38, pp. 1-11, 2003.

[13] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein, "Entirely declarative sensor network systems," in *Proceedings of the 32nd international conference on Very large data bases* Seoul, Korea: VLDB Endowment, 2006.

[14] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* San Diego, California, USA: ACM, 2007.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: a Tiny AGgregation service for ad-hoc sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 131-146, 2002.

[16] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*: Prentice Hall, 2007.

[17] Google, "Designing a Remote Interface Using AIDL," 2010.

[18] Maemo.org, "dbus guide for Maemo," 2010.

[19] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *Proc. ACM MobiSys*, 2010.

[20] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* San Jose, CA, 2002, pp. 85-95.

[21] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems* Boulder, Colorado, USA: ACM, 2006.

[22] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proc. ACM SIGOPS Symp. Operating Systems Principles (SOSP)* Big Sky, Montana, USA: ACM, 2009.

[23] freedesktop.org, "Software/dbus," 2010.

[24] J. J. Labrosse, *MicroC OS II: The Real Time Kernel*: Newnes, 2002.

[25] M. Raju, "TI Application Notes: Heart-Rate and EKG Monitor Using the MSP430FG439," 2007.

[26] J. Scarlett, "Analog Devices Application Notes: Enhancing the Performance of Pedometers Using a Single Accelerometer " 2007.

[27] "Shimmer - Wireless Sensor Platform for Wearable Applications," Shimmer Research, 2010.

[28] R. Chakravorty, "A Programmable Service Architecture for Mobile Medical Care," in *Proceedings of the 4th annual IEEE international conference on Pervasive Computing and Communications Workshops*: IEEE Computer Society, 2006.

[29] D. Malan, T. Fulford-jones, M. Welsh, and S. Moulton, "CodeBlue: An ad hoc sensor network infrastructure for emergency medical care," in *International Workshop on Wearable and Implantable Body Sensor Networks*, 2004.

[30] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta, "Somniloquy: augmenting network interfaces to reduce PC energy usage," in *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)* Boston, MA, 2009.

[31] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins, "Turducken: hierarchical power management for mobile devices," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* Seattle, WA, 2005, pp. 261-274.

[32] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff, "Tapping into the fountain of CPUs: on operating system support for programmable devices," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* Seattle, WA, USA: ACM, 2008.