A Generic Component-based Approach for Programming, Composing and Tuning Sensor Software

Amirhosein Taherkordi, Frédéric Loiret, Romain Rouvoy, Frank Eliassen

OXFORD UNIVERSITY PRESS The Computer Journal

A Generic Component-based Approach for Programming, Composing and Tuning Sensor Software

Journal:	*The Computer Journal*
Manuscript ID:	COMPJ-2010-08-0332.R1
Manuscript Type:	Original Article
Date Submitted by the Author:	06-Nov-2010
Complete List of Authors:	Taherkordi, Amirhosein; University of Oslo, Informatics Loiret, Frederic; INRIA Lille – Nord Europe Rouvoy, Romain; INRIA Lille – Nord Europe Eliassen, Frank; University of Oslo, Informatics
Key Words:	Wireless Sensor Networks, High-level Programming, Component Model, Event-driven

A Generic Component-based Approach for Programming, Composing and Tuning Sensor Software

Amirhosein Taherkordi†, Frédéric Loiret‡, Romain Rouvoy†‡, and Frank Eliassen†

†University of Oslo, Department of Informatics, Oslo, Norway

‡INRIA Lille – Nord Europe, ADAM Project-team, University Lille 1, LIFL CNRS UMR 8022, Villeneuve d'Ascq, France

Email: amirhost@ifi.uio.no, frederic.loiret@inria.fr, romain.rouvoy@inria.fr, frank@ifi.uio.no

Wireless Sensor Networks **(WSNs) are being extensively deployed today in various monitoring and control applications by enabling rapid deployments at low cost and with high flexibility. However, high-level software development is still one of the major challenges to wide-spread WSN adoption. The success of high-level programming approaches in WSNs is heavily dependent on factors like ease of programming, code well-structuring, degree of code reusability, required software development effort, and the ability to tune the sensor software for a particular application. Component-based programming has been recognized as an effective approach to satisfy such requirements. However, most of the componentization efforts in WSNs were ineffective due to various reasons, such as high resource demand or limited scope of use. In this article, we present Remora, a novel component-based approach to overcome the hurdles of WSN software implementation and configuration. Remora offers a well-structured programming paradigm that fits very well with resource limitations of embedded systems, including WSNs. Furthermore, the special attention to event handling in Remora makes our proposal more practical for embedded applications, which are inherently event-driven. More importantly, the mutualism between Remora and underlying system software promises a new direction towards separation of concerns in WSNs. This feature also offers a practical way to develop sensor *middleware services* which should be generic and developed close to the operating system. Additionally, it allows the customization of sensor software—deploying only application-required system-level services on nodes, instead of installing a fixed large system software image for any application. Our evaluation results show that the deployed Remora applications have an acceptable memory overhead and a negligible CPU cost compared to the state-of-the-art development models.**

Keywords: Wireless Sensor Networks; High-level Programming; Component Model; Event-driven

Received 00 Month 2010; revised 00 Month 2010

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are a rapidly emerging research area because of their vast application vistas in real-world environments. The advances in wireless communications and miniaturization of hardware components have enabled the development of low-cost, low-power, and multifunctional sensor nodes. These tiny devices can be easily embedded in the environment, establish a wireless ad-hoc network, and compose a distributed system to collaboratively sense and process the surrounding physical phenomenons as data. However, WSNs differ from the conventional distributed systems in many aspects. Resource scarceness is the most important uniqueness of WSNs.

Sensor nodes are often equipped with a limited energy source and a processing unit with a small memory capacity. Additionally, the network bandwidth is much lower than for wired communications and radio-based operations are the dominant energy consumer within a sensor node. The sensor nodes and network are less reliable than in conventional distributed systems. Depending upon the configuration of network and environment circumstances, wireless links may become degraded or unviable.

These factors make the way to develop WSN applications quite critical and also different from the other existing network systems. However, this concept is still immature in the context of WSNs for various reasons. Firstly, the existing diversities in WSN hardware and software platforms have brought the same order of diversity to programming models for such platforms [1]. Moreover, developers' expertise in state-of-the-art programming models become useless in WSN programming as the well-established discipline of program specification is largely missing in this area. Secondly, the structure of programming models for WSNs are usually sacrificed for resource usage efficiency, thereby, the outcome of such models is usually a piece of tangled code hardly maintainable by its owner. Finally, application programming in WSNs is mostly carried out very close to the operating system, forcing developers to learn low-level system programming models. This not only diverts the programmer's focus from the application logic, but also needs low-level programming techniques, which imposes a significant burden on the programmer.

From a software composition perspective, the way to implement WSN applications is also becoming increasingly important as today's sensor software not only consists of application and system modules, but also includes various off-the-shelf, third-party software products, such as middleware services. Ideally, such integrations should be realized through a meta-level abstraction with minimum programming effort. This, in fact, indicates the capability of a WSN programming model to facilitate the development of middleware services and their integration to target application software.

The ability to tune the sensor software for a particular use-case or application domain is the other major issue in this context. As sensor nodes are typically equipped with a limited memory capacity, operating system developers need to keep the size of system modules as small as possible in order to preserve enough memory space for application modules, and they also have to ensure the portability of system software to various sensor platforms. This mostly leads to software artifacts with either degraded functionality not satisfying all end-user expectations, or suffering from the lack of modularity and maintainability. One solution to tackle this problem is to consider the operating system as a collection of well-defined services deployable on a minimized kernel image so that the programmer has the ability to involve only application-required system services in the process of software installation. Therefore, this can bring a significant efficiency to resource usage in sensor nodes by avoiding installing a single monolithic operating system for any application.

Software *componentization* has been recognized as a well-structured programming model able to tackle the above concerns. Component-based programming provides an high-level programming abstraction by enforcing interface-based interactions between system modules and therefore avoiding any hidden interaction via direct function call, variable access, or inheritance relationships. This abstraction rather offers the capability of black-box integration of modules in order to simplify configuration and maintenance of software systems. Module reusability and provision of standard API are some other advantages of adopting *component-based software development* [2, 3]. Although using this paradigm in earlier embedded systems was relatively successful [4, 5, 6, 7], most of the efforts in the context of WSNs remain inefficient or limited in the scope of use. The TinyOS programming model, named NesC [8], is perhaps the most popular component model for WSNs. Whereas NesC eases WSN programming, this component model remains tightly bound to the TinyOS platform. Other proposals, such as OpenCom [9] and THINK [10], are either too heavyweight for WSNs, or not able to support event-driven programming, which is of high importance in WSNs.

In this article, we present extended results on Remora, a lightweight component model designed for resource-constraint embedded systems, including WSNs [11]. The strong abstraction promoted by this model allows a wide range of embedded systems to exploit it at different software levels from *Operating System* (OS) to application. To achieve this goal, Remora provides a very efficient mechanism for event management, as embedded applications are inherently event-driven. Remora components are described in XML as an extension of the *Service Component Architecture* (SCA) model [12] in order to make WSN applications compliant with the state-of-the-art componentization standards. Additionally, the C-like language for component implementation in Remora attracts both embedded system programmers and PC-based developers to programming for WSNs. Remora also features a coherent mechanism for component *instantiation* and *property-based component configuration* in order to facilitate lightweight event-driven programming in WSNs. Notably, in this paper the aforementioned features of Remora are extended in the following ways. First, we propose a programming approach, based on the concept of *Autonomous Composable Module* (ACM), to achieve a practical and efficient way of

developing component-based *middleware systems* in WSNs. Second, we introduce a mechanism to enable *tuning* system software by componentizing the OS-level services and customizing OS functionality based on target application's requirements. The REMORA specifications and their implementation techniques are also extensively explored in this paper.

As a matter of validation, we demonstrate the comprehensive evaluation results of deploying REMORA components on Contiki—a leading operating system for WSNs [13]. Specifically, we extend our earlier evaluation efforts in [11] with considering a complementary set of performance figures, such as required programming effort. The efficient use of Contiki features, such as process management and event distribution [14], on the one hand, and the abstraction layer linking REMORA to Contiki, on the other hand, promise a very effective and generic approach towards practical high-level programming in WSNs. In particular, we present the functionality of REMORA within the context of a real use case involving a network-level *application suite* in order to support *code distribution* in dynamic sensor applications. Finally, the evaluation work is completed by carrying out a comprehensive investigation of existing software component models for WSNs and comparing them with REMORA.

The remainder of this article is therefore organized as follows. In Section 2, the specification of the REMORA component model is presented. Section 3 describes how REMORA is implemented, while the evaluation results are reported in Section 4, including the assessment of a real REMORA-based deployment. A survey of existing approaches and a discussion on REMORA extension opportunities are presented in Section 5 and Section 6, respectively. Finally, Section 7 concludes this paper and identifies some future work.

2. REMORA COMPONENT MODEL

In this section, we first discuss the primary design concepts in REMORA and then we explain the specifications of the REMORA component model. The first obvious principle is that WSN applications in our approach are built out of components conforming to the REMORA component model. The other design principles of REMORA include:

XML-based Component Description. The first design goal emphasizes simplicity and generality of the technique for describing REMORA components. In REMORA, we therefore adopt XML technologies to describe components. The basis for the XML schema we defined is the *Service Component Architecture* (SCA) notations in order to provide a uniform component model covering components from sensors to the Internet, as well as to accelerate standardization of component-based programming in WSNs. As SCA was originally designed for large-scale systems-of-systems [12], REMORA extends SCA with its own architectural concerns to achieve realistic component-based programming in WSNs.

C-like Language for Component Implementation. REMORA components are written in a C-like language enhancing the C language with features to support component-based and structured programming. The other objective in this enhancement is to attract both embedded systems programmers and PC-based developers towards high-level programming in WSNs.

OS Abstraction Layer. The REMORA component framework is integrated with the underlying operating system through a well-defined OS-abstraction layer. This thin layer can be developed for various WSN operating systems supporting the C language, such as Contiki. This feature ensures the portability of REMORA components towards different OSs. The abstraction provided by REMORA becomes more valuable when the component framework is easily configured to reuse OS-provided features, such as event processing and task scheduling.

Event Handling. Event-driven programming is a common technique for programming embedded systems as memory requirements in this programming model is very low. Besides the support for events at the operating system level in embedded systems, we also need to consider event handling at the application layer. REMORA therefore proposes an high-level support of event generation and event handling, which makes it one of the key features of our proposal. In particular, REMORA achieves this goal by reifying the concept of event as a first-class architectural element simplifying the development of event-oriented scenarios.

Before describing our component model, we first define the basic terms used throughout this article. Figure 1 illustrates the development process of REMORA-based applications. A REMORA application consists of a set of REMORA Components, containing descriptions and implementations of software modules. The REMORA *engine* processes the components and generates standard C code deployable within the REMORA *framework*. The framework is an OS-independent module supporting the specification of the REMORA component model. Finally, the REMORA application is deployed on the target sensor node via the REMORA *runtime*, which is an OS-abstraction layer integrating the application to the system software.

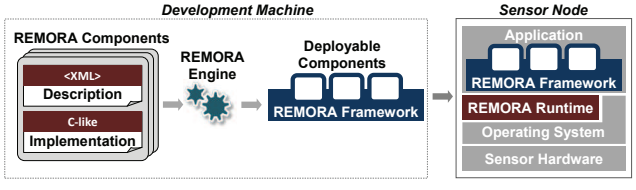

FIGURE 1. Development process of REMORA-based applications.

2.1. Component Specification

A REMORA component contains two main artifacts: *component description* and *component implementation*. The component description is an XML document containing the specifications of the component including its *services*, *references*, *producedEvents*, *consumedEvents*, and *properties* (cf. Figure 2). A *service* can expose a REMORA interface, which is a separate XML document specifying the functions provided by the component, while a *reference* indicates the operations required by the component as an interface. Likewise, a *producedEvent* identifies an event type generated by a component, whereas a *consumedEvent* specifies component's interest on receiving a particular event. The component implementation is a C-like program containing three types of operations: *i)* operations implementing the component's services, *ii)* operations processing events, and *iii)* component's private operations.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType name="COMPONENT_NAME">
  <service name="SERVICE1_NAME">
    <interface.remora name="INTERFACE1_NAME"/>
  </service>
  ... other services
  <reference name="REFERENCE1_NAME">
    <interface.remora name="INTERFACE2_NAME"/>
  </reference>
  ... other references
  <property name="PROP1_NAME" type="PROP1_TYPE">
    PROP1_DEFAULT_VALUE
  </property>
  ... other properties
  <producer>
    <event.remora type="EVENT1_TYPE" name="EVENT1_VAR_NAME"/>
  </producer>
  ... other producers
  <consumer operation="CONSUMER_OPERATION">
    <event.remora type="EVENT2_TYPE" name="EVENT2_VAR_NAME"/>
  </consumer>
  ... other consumers
</componentType>
```

FIGURE 2. The XML template for describing REMORA components.

To make the specification more concrete, we first present a simple example of a REMORA-based application, then we discuss REMORA features carefully. This simple application is in charge of *blinking* a LED on a sensor node every three seconds. Figure 3 depicts the components involved in this application.

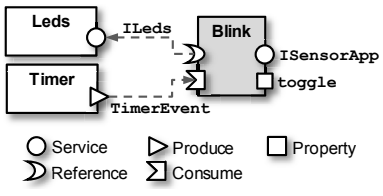

FIGURE 3. A simple REMORA-based application.

We here focus on the Blink component and describe it according to the REMORA component model. In Figure 4, the XML description of the Blink component is shown. This component provides an `ISensorApp` interface to start application execution and requires an `ILeds` interface to switch LEDs on and off, which is implemented by the Leds component. It also owns a property to `toggle` a LED on the sensor node. As the Blink component produces no event, the `producer` tag in the component description is empty, while it is subscribed to receive TimerEvent and process this event in the `timerExpired` function. The last part of the component description is the libraries used by the component implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType name="app.BlinkApp">
  <service name="iSensorApp">
    <interface.remora name="core.boot.api.ISensorApp"/>
  </service>
  <reference name="iLeds">
    <interface.remora name="core.peripheral.api.ILeds">
  </reference>
  <property name="toggle" type="xsd:short">0</property>
  <producer/>
  <consumer operation="timerExpired">
    <event.remora type="core.sys.TimerEvent" name="aTimeEvent"/>
  </consumer>
  <libraries>
    <include name="stdio" type="SystemLib"/>
  </libraries>
</componentType>
```

FIGURE 4. XML description of Blink component.

Figure 5 presents the excerpt of the Blink implementation. This C-like code implements the single function of the `ISensorApp` interface (`runApplication`) and handles TimerEvent within the `timerExpired` function. In the `runApplication` function, we specify that the TimerEvent generator (`aTimeEvent.producer`) is configured to generate periodically TimeEvent every three seconds. The last command in this function is also to notify the TimerEvent generator to start time measurement. When time is expired, Timer sets the attributes of aTimeEvent (*e.g.*, latency) and then the REMORA framework calls the `timerExpired` function.

```
void runApplication(){
  printf("--- Starting Blink Application ---");
  short periodic = 1;
  aTimeEvent.producer.configure(3*CLOCK_SECOND, periodic);
  aTimeEvent.producer.start();
}
void timerExpired(){
  if (this.toggle == 0){
    iLeds.onLeds(LEDS_RED);
    this.toggle = 1;
  }else{
    iLeds.offLeds(LEDS_RED);
    this.toggle = 0;
  }
  printf("Time elapsed after interval: %d", aTimeEvent.latency);
}
```

FIGURE 5. C-like implementation of Blink component.

Services and References. The first step towards component-based programming is identifying system services, and then identifying which component(s) provides a service and which one(s) requires the service (so called reference). Similar to component descriptions in REMORA, interfaces are described in XML. Interface description includes a name and the associated operations. Figure 6 presents the simplified `ILeds` interface used by the Blink component as a reference. Every component providing a service should implement all the operations specified in the interface description with the same signatures.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface.remora name="core.peripheral.api.ILeds">
  <operation name="getLeds" return="xsd:unsignedByte"/>
  <operation name="onLeds">
    <in name="leds" type="xsd:unsignedByte"/>
  </operation>
  <operation name="offLeds">
    <in name="leds" type="xsd:unsignedByte"/>
  </operation>
</interface.remora>
```

FIGURE 6. A simplified description of `ILeds` interface.

Component Properties. In REMORA, programmers can define properties for a component. Properties enable reconfiguration of component behaviors and also convert components from a dead unit of functionality to an active entity tractable during the application lifespan. The component reconfiguration becomes very essential for event producer components, *e.g.*, to generate accurate TimerEvents in the Blink application, we need to configure the Timer component through a property that holds the time at which the measurement is started. Properties also enable components to become either *stateless* or *stateful*. A component is stateful if and only if it defines a property—*e.g.*, the Blink component in our sample application is a stateful component retaining the value of the `toggle` property—whereas the Leds component is a stateless component. The properties of a component can be accessed from the component implementation using the keyword `this`.

Component Implementation. REMORA components are implemented by using a dialect of C language with a set of new commands. This C-like language is mainly proposed to support the unique characteristics of REMORA, namely, component instantiation, event processing, and property manipulation. Therefore, for pure component-based programming without the above features, the programmer can almost rely on C features and develop an elementary REMORA-based application including only REMORA-based interface invocations. We implicitly introduced a few of these commands within the Blink component implementation, while the complete description of commands is available in [15].

Parameter-based Reconfiguration. To preserve efficiency in resource usage, REMORA relies on compile-time linking so that system components are linked together statically and their memory address is also computed at compile-time. Additionally, for multiple-instance components, all required instances are created in compiler-specified addresses prior to application startup. These constraints not only reduce the size of the final code, but also relieve the programmer from the burden of managing memory within the source code. In REMORA, the reconfiguration feature is also considered from a parametric perspective: A REMORA component can be reconfigured statically by changing the behavior of its functions through its component *properties*. In particular, for the property-dependent functions of a component, the behavior of the component can easily be changed by adjusting property values and thus a form of parameter-based reconfigurability is enabled within the component.

2.2. Component Instantiation

REMORA features a concrete mechanism to support component instantiation. This feature is essentially proposed to manage efficiently event producer components. The REMORA engine greatly benefits from component instantiation when undertaking linking of one event producer to several consumer components. For example, in the Blink application, the producer (Timer) of TimerEvent should be instantiated per consumer component, while the UserButtonEvent generator is a single-instance component publishing an event to all subscribed components when the user button on a sensor node is pressed.

By component instantiation, we refer to two principles: *i)* the component code is always single-instance, and *ii)* the component *context* is replicated per instance. By component context, we mean the *data structures* required to handle the properties independently from the component's code. By doing that, a REMORA component becomes a *reconfigurable and reusable* entity with a strong abstraction, and more importantly the memory overhead is kept very low by avoiding code duplication.

REMORA proposes three *multiplicity types* for the component's context: *raw-instance* (stateless component), *single-instance*, and *multiple-instances*. The REMORA engine features an algorithm computing the multiplicity type of a component based on three parameters: *i)* whether the component owns any property, *ii)* whether the component is an event producer, and *iii)* the number of components subscribed to a specific event. When the multiplicity type is determined, the REMORA engine statically allocates memory to each component instance.

2.3. Event Management

As high-level event processing is a necessary functionality in embedded systems, the REMORA design comprehensively supports events between components. The main goal is to reify the concept of event as a first-class architectural element simplifying the development of event-oriented scenarios at a low cost. The event design principles in REMORA include:

Event Attributes. An event type in our approach can have a set of attributes with specific types. By defining attributes, the event producer can provide the event-specific information to the event consumer, *e.g.*, the `latency` attribute of TimerEvent in the Blink application.

Application Events vs OS Events. Events in our framework are categorized into two classes: *application-events* and *OS-events*. Application-level events are generated by the REMORA framework (like Timer in

the Blink application), while the latter are generated by the sensor operating system. In other words, the only difference of these two types is the source of event generation. To process OS-events at the application level, the Remora runtime features mechanisms to observe OS-events, translate them to corresponding application-level events, and publish them through *OS-event producer* components.

Event Observation Interface. One of the important aspects of event processing is the time period in which events should be observed by the event producer. Obviously, the length of this period varies with the type of events, *e.g.*, the observation period for a TCP/IP event is the whole application lifespan (*automatic* observation), while a Timer event is observed according to the user-configured time (*manual* observation). Remora therefore proposes the *event observation interface* in order to control the manual observations. This interface includes event control operations, such as `start`, `pause`, `resume`, and `terminate`. If an event type is manually observable, the associated event producer should implement the generic observation interface. By doing that, the event consumer can handle the life cycle of the observation process by calling the operations of this interface without being aware of the associated event producer.

Event Configuration Interface. The specification of an event type in our approach contains a *configuration interface*. Each component producing an event should implement the associated configuration interface. This feature enables the event consumer to configure event generation before starting the event observation process. More importantly, by introducing such an interface within the event specification, the event producer and the event consumer become completely decoupled, *e.g.*, in the Blink application, TimerEvent generation is configured within the Blink component without being aware of the associated event generator.

Single Event Producer per Event Type. Each event type in our approach is produced by *one and only one* component. Instead of imposing the high overhead of defining event channels and binding event consumers and producers, we ease event-based programming by assuming one-to-one association between event types and event producers. The programmer is also released from identifying such bindings as the Remora framework becomes responsible to automatically wire producers and consumers. We believe that this assumption does not affect event-related requirements of embedded platforms. In case an event is produced by two different components, the programmer can define a new event type, extended from the original event, for one of the producer components.

2.3.1. Event Casting

Events in our proposal can be either *unicast*, or *multicast*. Unicast is a one-to-one connection between an event producer and an event consumer—*e.g.*, TimerEvent in the Blink application. In contrast to the unicast model, a multicast event may be of interest to more than one component—*e.g.*, a UserButtonEvent may be handled by several components. The Remora framework distinguishes between these two types in order to improve the efficiency of processing and distributing events. Event distribution should also be considered together with component instantiation. We need to clarify how multiplicity type of components on the one side, and unicast events and multicast events on the other side are related. To this end, we define two invariants:

> Invariant1: *The consumer of a unicast event should be a raw-instance or single-instance component.*
>
> Invariant2: *The producer of a multicast event should be a raw-instance or single-instance component.*

These invariants are mainly proposed to boost the efficiency of event processing in the Remora framework. We do not support other event communication schemes since it implies to reify at runtime the source and the destination of an event and to maintain complex routing tables within the Remora framework, which will induce significant overheads in term of memory footprints and execution time. We rather believe that these invariants do not limit event-related logic of embedded applications.

2.3.2. Events Description

Similar to components, events have their own descriptions, which are in accordance to the event specification in Remora. Figure 7 presents a simplified events description document of the Blink application. This document consists of two outer tags: `remora-events` and `os-events`, corresponding to the application level events and the OS events, respectively. For each event type, we can specify its observation model and casting type. The attributes of an event are also described by the attribute tag and the operations of event configuration interface is specified by the `configInterface` tag.

2.3.3. Event Management Illustration

Figure 8 illustrates the event management mechanism implemented in Remora. We explain the mechanism based on the steps labeled in the figure. During the first two steps, the event consumer can configure event generation and control event observation by calling the

```
<?xml version="1.0" encoding="UTF-8"?>
<events>
  <remora-events>
    <event-type name="core.sys.TimerEvent"
                castType="unicast" observation="manual">
      <attribute name="latency" type="xsd:int"/>
      <configInterface>
        <operation name="configure">
          <in name="interval" type="xsd:int"/>
          <in name="periodic" type="xsd:short"/>
        </operation>
      </configInterface>
    </event-type>
  </remora-events>
  <!-- add other application event types here -->
  <os-events>
    <!-- describe OS-events here -->
  </os-events>
</events>
```

FIGURE 7. Application events description.

associated interfaces realized by the event producer component. These steps in our sample application are achieved in the Blink component (event consumer) by the code below:

```
aTimeEvent.producer.configure(3*CLOCK_SECOND,
periodic);
aTimeEvent.producer.start();
```

Note that the programmer is not aware of the TimerEvent producer. She/he only knows that the TimerEvent generator is expected to implement the `configure` function defined in the description of TimerEvent (cf. Figure 8). The TimerEvent producer should also implement the observation interface as the observation type of TimerEvent is manual.

Whereas the above steps are initiated by the component programmer, the next two steps are performed by the REMORA component framework. Step 3 is dedicated to *polling* the producer component to observe event occurrence. The event producer is polled by the REMORA framework through a *dispatcher* function in the producer. In fact, the event observation occurs in this function. The polling process is started, paused, resumed, and terminated based on the programmer's configuration for the event observation, performed in step 2.

For application-level events, the REMORA framework is in charge of calling periodically this function, while for OS-events, REMORA invokes this function whenever an OS-event is observed by the REMORA runtime. The REMORA runtime listens only to application-requested OS-events, and delivers the relevant ones to the framework. The REMORA framework then forwards the event to the corresponding OS-event producer component by calling its dispatcher function—*e.g.*, `user_button` is a Contiki-level event that should be processed by the REMORA component UserButton. This component then generates an high-level UserButtonEvent and publishes it to the REMORA framework.

Finally, in step 4, upon detecting an event in the dispatcher function, the producer component creates the associated event, fills the required attributes, and publishes it to the REMORA framework. The framework in turn forwards the event to the interesting components by calling their event handler function.

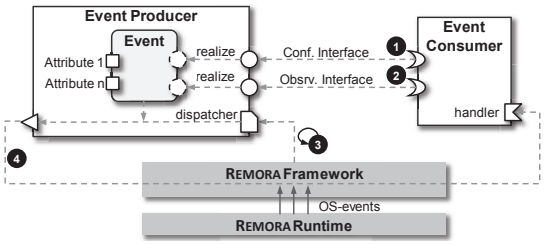

FIGURE 8. Event management mechanism in REMORA.

2.4. Components Assembly and Deployment

A typical REMORA application may contain several implementations of a given component type due to the existing heterogeneity in WSN hardware and software platforms. To configure an application according to the target platform requirements, REMORA introduces components *assembly* (equivalent to *composite* component in SCA). This XML document specifies the list of application components, as well as bindings between references and services of components. Figure 9 shows the configuration of Blink application in which there is only one binding from Blink to the Leds component implementing the interface `ILeds` for the MSP430 microcontroller. Note that, based on the event casting invariants, the event-binding between Blink and Timer is created automatically by the REMORA framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="app.BlinkAppConfigurer">
  <component name="ledControl">
    <implementation.remora
              implementer="cmu.telosb.peripheral.Leds"/>
  </component>
  <component name="blink">
    <implementation.remora implementer="app.BlinkApp">
  </component>
  <component name="timer"/>
    <implementation.remora implementer="core.sys.Timer"/>
  </component>
  <!—components wiring -->
  <wire source="blink/iLeds" target="ledControl/iLeds"/>
</composite>
```

FIGURE 9. Blink application configuration.

Figure 10 illustrates the four main phases of an application deployment. The REMORA development box encompasses artifacts supporting component specification. Events description and components configuration are used to describe system events and components assembly, respectively. Components and interfaces are also described in separate XML documents, one for each. External types are a set of C header files containing application's type definitions. The last group of elements in this box are C-like implementation files of components in which OS libraries may be called through a set of System APIs implemented by REMORA runtime components. Note that there is no hard-coded dependencies between REMORA implementers and the native API of the

underlying OS (*e.g.*, Contiki) to ensure portability of REMORA components towards different OSs.

In the next phase, the REMORA engine reads the elements of the development box and also OS libraries in order to generate the REMORA framework including the source code of components and OS-support code. Then, application object file will be created through OS-provided facilities and finally deployed on sensor nodes.

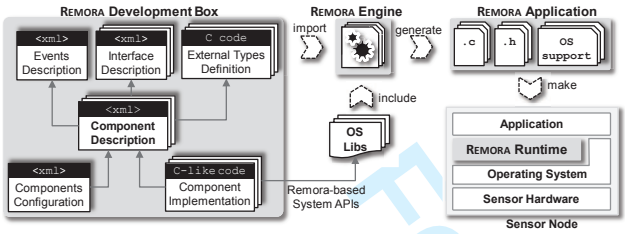

FIGURE 10. REMORA-based development process.

2.5. Middleware Programming

The research efforts on sensor middleware have hitherto focused on developing services and algorithms for routing, quality of service, energy-efficiency, resource management, localization, synchronization, etc. These, however, often fall short of expectation in integrating services and algorithms into a generic middleware system, and in helping application programmers to compose a system that exactly matches their requirements. This raises the need for a specific approach for middleware programming in WSNs that goes beyond dealing with only application-specific logics. From the programming point of view, middleware services are distinguished from other components in the system by the following two main factors.

First, despite the application-level programming, middleware components are developed very close to the operating system, requiring to tightly interact with system-level components. Therefore, sensor programming models, supporting middleware development, should provide the primitives required to interface between middleware services and system components. REMORA addresses this concern through the OS Abstraction Layer and the OS-Wrapper components. In addition to enabling the portability of sensor applications, these principles make REMORA a suitable programming model to build middleware applications.

Second, middleware solutions should be exposed as a well-packaged, stand-alone application which can be easy integrated to the target application with minimum programming effort. Although this issue has been extensively addressed in conventional resource-rich systems, software pieces in WSNs are often assembled together in an ad-hoc manner, without any well-established software composition model. This problem originates from the fact that WSN programming abstractions do not pay enough attention to software composition and integration approaches. With the increasing number of intermediate software solutions for WSNs (*e.g.*, networking, algorithms and QoS), programming constructs are required to compose the application, middleware services, and the operating system into a unified sensor software in a generic, simple and robust manner.

The technique we have adopted in REMORA to compile and assemble components has the potentials to meet a higher level of assembly which is *integrating a given set of* REMORA*-based applications*. In particular, we enhance the REMORA engine with the capability of processing multiple isolated REMORA applications and integrating them into a unified system. The main concerns, in this endeavor, include how to expose an application's functionality as an API and bind applications based on the dependencies between their APIs. REMORA addresses these concerns based on the concept of *Autonomous Composable Module* (ACM). This refers to developing REMORA applications in an autonomous manner so that the programmer considers an under-development application as a stand-alone module with its own operations. It means that, based on this approach, the dependencies of the application to others are not declared within its description. The REMORA engine is in charge of analyzing dependencies among ACMs and binding them together. Figure 11 shows the overall architecture of REMORA composition solution, consisting of a set of ACMs and the main sensor application. The latter not only implements the application logic, but also serves as a starting point to execute programs. An ACM contains a set of REMORA components implementing its logics, as well as a component representing its API.

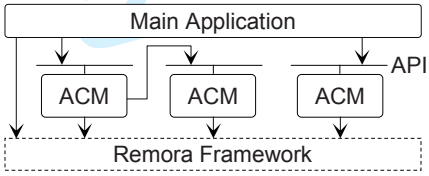

FIGURE 11. The overall architecture for composing the main application and ACMs.

As a use case for the REMORA middleware programming model, in [15] we demonstrate a run-time middleware system, called REMOWARE, to support dynamic reconfiguration of REMORA-based applications. REMOWARE is basically an ACM which can be easily used as a middleware solution in any dynamic sensor application to enable run-time reconfiguration of REMORA components.

2.6. Automatic Tuning

Besides the componentization of application-level modules, REMORA can be exploited to componentize operating system's modules either by wrapping them in REMORA components, or redeveloping them according to the REMORA specification. This enables the REMORA engine to expand its control on the configuration of sensor software and therefore makes it possible to automatically tune the target software installed on nodes. In this way, the REMORA engine can gain a meta knowledge showing which OS-level components are involved in supporting application logic and based on that it can trace the interactions between application components and system components. In this way, it can identify the *orphan* components—the components that are not involved in the application scenario execution.

Figure 12 describes an *initial* configuration (prior to deploying on nodes) of sensor software in which the application-level components gain system services through OS-wrapper components at the runtime layer. These components interact either directly with kernel-level modules, or with other intermediary wrapper components beneath the runtime. This initial setting can be optimized by REMORA engine. When it executes the tuning process, deduces that one of the intermediary components is orphan, and removes it from the final package installed on nodes.

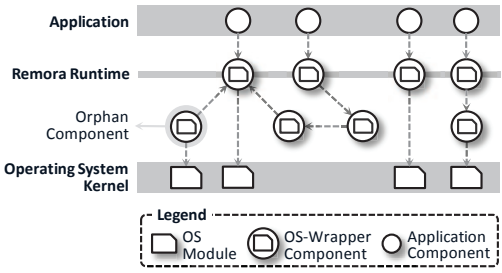

FIGURE 12. The REMORA engine tunes the operating system by tracing component dependencies and finding orphan components.

3. IMPLEMENTATION

To discuss the implementation of REMORA, we structure this section according to the main modules proposed for REMORA-based application development, namely, the *engine*, the *framework*, and the *runtime*. Since the platform supporting the component model is comprehensive and includes numerous implementation issues, we only highlight the key technologies and design techniques used for implementing each of the aforementioned modules. Beyond the internal design of modules, the overall design goal is to keep the artifacts of each module completely independent from others in the sense that in the final system, each module is composed of three set of source codes dedicated to corresponding modules. The main advantage of this separation is to minimize the required effort to port the component model to a new operating system by ensuring a clear isolation between the REMORA framework and the REMORA runtime.

3.1. Remora Engine

The REMORA engine is deployed on the programmer's desktop machine to read all artifacts within the development box, perform required analyses for code generation, and generate the final C code of components, as well as OS-support code. We adopt Java to develop the engine because of its cross-platform capabilities, as well as its strong support for XML processing. Additionally, the object-oriented nature of Java simplifies the complex process of code analyzing and code generation. We briefly discuss the key design principles of this Java-based engine below.

The first task of the engine is to parse the C-like implementation of components and extract the information concerning the specification of REMORA. To this end, we have developed a parser module, which is originally generated by ANTLR—a widely used open-source parser generator [16]. Since this generated tool only parses the source code, we have modified the generated parser to extract REMORA-required information, such as *name*, *signature*, and *body* of implementation functions. By doing that, the engine builds a meta-data structure containing all required information about the implementation of a component and the rest of the engine tasks are performed based on that.

The other key implementation part of the REMORA engine deals with *processing events*, *component instantiation*, and *component lifecycle*. This unit deduces the multiplicity type of components according to the algorithm 1 and generates the necessary data structures. This algorithm determines the multiplicity type based on the type of events generated by the component, as well as whether the component owns any property or not. If the final value of variable $InstNumber$ is 0, this means that the component has no instance and only requires the code memory, while the value of 1 shows that only one instance of component's data should be stored in the data memory. Finally, for a multiple instance component the value of $InstNumber$ is 2.

This module also features a set of well-defined techniques, such as *in-component call graph analyzer* and *cross-component call tracker* to support stateful component. The former concept is concerned with discovering *state-dependent* functions of a component. Two types of state dependency can be envisaged for a function: *i) explicit dependency*: the component's property(s) is(are) directly accessed within the function's code, *ii) implicit dependency*: the function contains direct/indirect invocation(s) to an explicit type. To preserve the state of a component, we need to retain

Algorithm 1 Determining the multiplicity type of components

Input: *producedEvents*, events generated by the components
Input: *properties*, component's properties
Output: component's multiplicity type

```
InstNumber ⇐ −1
MultiConsumers ⇐ false
for aEvent in producedEvents do
  if aEvent is unicast then
    if sizeOf(aEvent.consumers) > 1 then
      MultiConsumers ⇐ true
      break
    end if
  end if
end for
if MultiConsumers is false then
  if sizeOf(producedEvents) > 0 then
    InstNumber ⇐ 1
  else
    if sizeOf(properties) > 0 then
      InstNumber ⇐ 1
    else
      InstNumber ⇐ 0
    end if
  end if
else
  InstNumber ⇐ 2
end if
```

a pointer to the component's context and pass it to the state-dependent functions of component. The in-component call graph analyzer employs a recursive technique to navigate the function calls with the component and identify the state-dependent functions. Likewise, the cross-component call tracker tracks the interactions between components in order to retain the state of components. Finally, the major task of the engine is to support events and manage the component lifecycle by embedding framework-support patches in the component implementation.

Automatic tuning of sensor software is the other responsibility of the REMORA engine. The data structure supporting the tuning process is a directional graph in which every node represents a component of the system and edges between nodes are the service-based interactions among the components (cf. Figure 13). The engine first creates this graph and then navigates the nodes based on the Depth-First Search (DFS) algorithm to find the orphan nodes. In particular, it initiates this process from the main component of application, implementing the interface `ISensorApp`, as the root of graph. When it accomplishes DFS, it removes orphan nodes—all components that are never visited by DFS.

Moreover, the REMORA engine undertakes binding ACM modules in order to support middleware

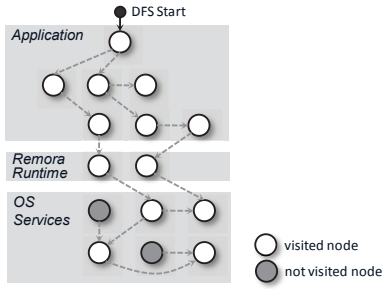

FIGURE 13. Using depth-first search algorithm to discover the orphan nodes.

programming. This process is carried out in a two-phase strategy. It first processes the components configuration document of each ACM and creates a disconnected, directed graph structure in which each ACM would have directed edges to the required APIs. In the second phase, the engine analyzes the yielded disconnected graph from the first phase and creates a connected graph representing dependencies among ACMs, as well as between the main sensor application and ACMs. Therefore, it provides a higher-level of wiring model between co-habiting applications and this model is further processed by the engine to implement the execution flow graph in the system.

3.2. REMORA Framework

The REMORA framework is composed of a collection of core C programs, supporting the event management model of REMORA and hosting the target application's components. As mentioned before, the REMORA framework is an OS-independent module. There are two main reasons for this: *i)* the core of the framework is written in the C language and also the final code of application's components are translated to equivalent C programs by the REMORA engine, *ii)* the framework is linked to the OS via the REMORA runtime which translates all OS-originated interactions (*e.g.*, OS-events) to a set of pre-defined, application-specific instructions understandable by the framework (cf. Section 2.3). The other possible dependency issue is caused by the mechanism used to form the REMORA framework as a *process* within the OS and *schedule* it to run. This is also extensively addressed by the REMORA runtime as explained in Section 3.3.

The main mission of the framework is to facilitate event management tasks, including *scheduling* and *dispatching*. To explain these tasks, we first introduce two *queue* data structures supporting our event model. The first queue is dedicated to the *event producer* components (PQ), while the second one is designed to maintain the *event consumers* (CQ). We discuss here how the REMORA framework is built based on these data structures.

Scheduling in REMORA refers to all operations required to *enqueue* and *dequeue* event producers and

event consumers. In particular, the main concern is *when* to enqueue/dequeue a component and *who* should perform these tasks. The REMORA framework addresses these issues based on the observation model of events. For example, if an event is *automatically* observable, the associated producer component and all the subscribed consumers are enqueued by the framework core during the application startup, while in a *manual* observation, producer and consumer are placed respectively in PQ and CQ when the consumer component calls the `start` function of observation interface. A question may arise is that prior to initiating the scheduling mechanism, how the components instances are created. In REMORA, memory allocation for components is done statically. Therefore, the memory address of all instances of all components are determined during the framework compilation and we do not impose the high overhead of dynamic memory allocation to such a resource-constraint platform. At runtime, parts of the framework, embedded in each component, are responsible for dealing with component lifecycle—*e.g.*, activating or deactivating event generator components.

The other role of the REMORA framework is to periodically poll the generator components for event observation, and then feed event handlers with the matched events. To achieve the former, event generators in REMORA keep a pointer to the globally known callback function, *dispatcher*, thereby, the REMORA framework is able to poll event generators by periodically calling this function. Similarly, the latter is realized by invoking the callback handler function within the event consumer component like `timerExpired` in the Blink component.

Figure 14 illustrates the *dispatching* mechanism in the framework including the supporting data structures. In *Polling*, the REMORA framework continuously polls the EventProducer components through *dispatcher*—the globally known callback function. Whenever a producer dispatches an event (AbstEvent), the framework casts this event to the actual event type, which is either UCastEvent(unicast event) or MCastEvent(multicast event). UCastEvent will be directly forwarded to the subscribed consumer through the callback function pointer stored in the UCastEvent. If a MCastEvent is generated, the framework delivers it to all the interesting components formerly enqueued. For OS-events, the same procedure is followed except the polling phase, which is performed by the operating system.

3.3. REMORA Runtime

The REMORA framework is integrated with the underlying operating system through the REMORA runtime. In our current implementation, the core of the REMORA runtime is a Contiki-compliant *process* running together with all other *autostart* processes

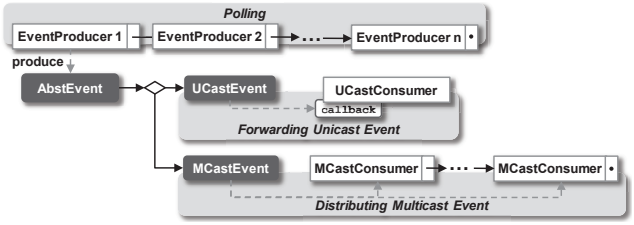

FIGURE 14. REMORA event processing mechanism.

of Contiki (see Figure 15). This process undertakes two tasks: *i)* periodically scheduling the REMORA framework (for polling event generator components) to run, and *ii)* listening to the OS-events and delivering the relevant ones to the REMORA framework. By relevant, we mean the REMORA runtime recognizes those OS-events that are of interest to the application. To achieve such a filtering, the source code of this part is generated by the REMORA engine according to the events description (cf. Section 2.3.2) of target application and then imported to the REMORA runtime. By doing that, we provide a lightweight event dissemination mechanism interpreting only application-specific OS-events.

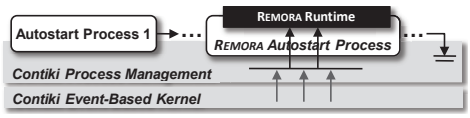

FIGURE 15. Integration of Contiki and REMORA through the runtime layer.

In addition, the application code may need to use libraries available in the OS. In REMORA, a programmer can develop a set of REMORA components acting as system API providers. In fact, these components delegate all high-level system calls to the corresponding OS-level functions—*e.g.*, the `currentTime()` function call in the system API is delegated to the Contiki function `clock_time()`. We offer this API to decouple the application components from OS modules and ensure the portability of REMORA-based applications. If an application is not expected to be ported to other operating systems, programmers can directly call the OS functions within component code and therefore slightly improve the runtime performance.

4. EVALUATION

To evaluate the efficiency of REMORA, in this section we first demonstrate and assess a real REMORA-based application, then we focus on the general performance figures of REMORA.

4.1. A Real Remora-based Deployment

Our real application scenario is a network-level *application suite* consisting of a set of mini applications

bundled together. This suite is basically designed to provide services, such as *code propagator* and *web facilities* in WSNs. We focus here on the first one and design it based on the REMORA approach.

Code propagation becomes a very important need in WSNs when we need to update remotely the running application software [17]. The code propagator application is responsible for receiving all segments of a running application's object code over the network and loading the new application image afterwards. The code propagator exploits the TCP and UDP protocols to propagate code over the network. At first, TCP is used to transfer new code, block by block, to the sink node connected to the code repository machine, and then UDP is used to broadcast wirelessly new code from a sink node to other sensor nodes in the network. When all blocks are received, the code propagator loads the new application.

Figure 16 describes the components involved in the first part of our application scenario. TCPListener is a core component listening to TCP events. This multiple-instances event generator is created for each TCP event consumer component with unique listening port number. For example, CodePropagator receives data from port 6510 (`codePropPort`), while WebListener is notified for all TCPEvents on port 80 (`webPort`). CodePropagator stores all blocks of new code in the external flash memory through the interface `IFile` implemented by the FileSystem component. When all blocks are received, CodePropagator loads the new application by calling the interface `ILoader` from the component ELFLoader. These two interfaces are system APIs that delegate all application-level requests to the OS-specific libraries. The interface `INet`, implemented by the component Network, is also the other system API providing the low-level network primitives to TCPListener.

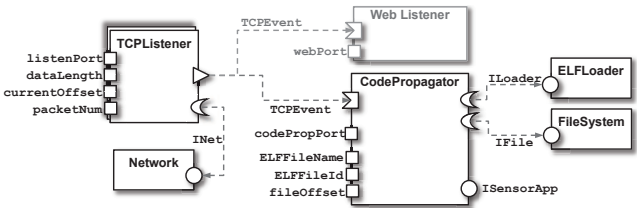

FIGURE 16. Code propagation application architecture.

As mentioned before, we adopt Contiki as our OS platform to assess the REMORA component model. Contiki is being increasingly used in both academia and industrial applications in a wide range of sensor node types. Additionally, Contiki is written in the standard C language and hence REMORA can be easily ported to this platform. Finally, the great support of Contiki on event processing and process management motivate us to design and implement the REMORA runtime on this OS. Our hardware platform is the popular TelosB mote equipped with a 16-bit TI MSP430 MCU with $48KB$ ROM and $10KB$ RAM.

TABLE 1. The memory requirement of code propagation application in REMORA-based and Contiki-based implementations.

Programming Model		Code Memory (bytes)	Data Memory (bytes)
Contiki		**722**	**72**
REMORA	Code Propagation Components		
	CodePropagator	252	36
	TCPListener	310	0
	System API Components		
	ELFLoader	38	0
	Network	92	0
	FileSystem	68	0
	REMORA Core		
	Framework and Runtime	494	14
	Total	**1254**	**50**
Remora overhead		**+532**	**-22**

The concrete separation of concerns in this application is the first visible advantage of using REMORA. The second improvement is the *easy* reuse of TCPListener for other TCP-required applications, which is not the case in a non-componentized implementation. In particular, for each new application, we only need to instantiate the *context* of TCPListener and configure its properties (like port number) accordingly—*e.g.*, WebListener in Figure 16.

Memory Footprint. Table 1 reports the memory requirement of REMORA and Contiki programming model (*protothreads*) for implementing the code propagation application. As indicated in the table, the REMORA-based development does not impose additional data memory overhead, while it consumes extra 532 bytes of code memory, which is essentially related to the cost of framework and runtime modules. This cost is paid once and for all, regardless of the size and the number of applications running on the sensor node. The code memory cost could be even further reduced by removing system APIs (Network, FileSystem, and ELFLoader) and calling directly the Contiki's libraries within CodePropagator. Note that the overhead of TCPListener can also be decreased when this component is shared for the use of other applications—*e.g.*, WebListener. Therefore, we can conclude that the memory overhead of REMORA is negligible compared to the high-level features it provides to the end-user.

Processing Cost. Figure 17 reports the comparison of CPU costs in these two approaches. The time measurement was started when the first block of new application's code was received and it was stopped when the last block of code arrived to the sensor node. Since in-file seeking and writing is a costly process, we removed invocations related to FileSystem and ELFLoader and measured the execution time afterwards. As the size of new code (ELF file)

TABLE 2. Line of code for our main components.

Component	Line of Code (Remora)	Line of Code (Contiki)	Reduced Effort
TCP Listener	62	104	**41%**
Code Propagator	19	36	**47%**

is increased, the processing overhead of REMORA is also slightly increased compared with the equivalent Contiki implementation. We believe that this very low overhead is due to the extra context-switchings (among event processing functions within the REMORA runtime) occurring for larger code in REMORA, which is not the case in the Contiki-based implementation.

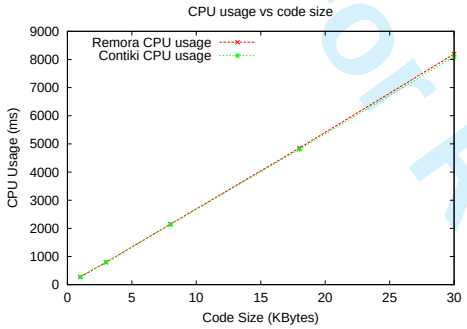

FIGURE 17. CPU usage for receiving new code by propagator application in REMORA and Contiki.

Programming Effort. Evaluating the programming effort is difficult since it is affected by factors difficult to measure—*e.g.*, the nature of code (algorithmic or routine), the complexity of the processing, and syntax and semantic of programming languages. However, WSN programming research has hitherto adopted the number of *lines of code* (LOC) as a simple indication. Table 2 reports this metric for the two main components of code propagator application. It is interesting to compare these measurements against the equivalent functionality available in Contiki libraries, where it is directly developed atop of the operating system. The Contiki-based implementation of the TCP listener module contains 41% more LOC than our version. This efficiency is achieved since in our implementation event-handling code is embedded in the run-time system and shared for the use of different applications. We also gain a significant improvement in LOC for code propagator module compared with the Contiki's implementation. It is because the verbose code of event handling in Contiki programming model is replaced with the shortened C-like code of REMORA.

Tuning Result. The efficiency of the tuning technique directly depends to the target use case and its requirements in terms of low-level system services. In the case of the code propagation application, we cannot precisely measure the reduction of the final object code size as it is basically an intermediate application lying beneath the main sensor application. Therefore, we measure the tuning performance of the code propagator by considering it as a main sensor application. Applying tuning technique on this application yields 5% reduction in the final Contiki binary object file. This efficiency is achieved by automatic removal of modules that never involve in the code propagation process, *e.g.*, programs interfacing a node's peripherals (*e.g.*, light, button and sensors).

TABLE 3. The minimum memory requirement of REMORA.

Module	Code Memory (bytes)	Data Memory (bytes)
Framework Core	374	4
Runtime Core	120	10
Total	**494**	**14**

The rest of this section is devoted to the assessment of two main performance figures of REMORA, namely, memory footprints and CPU usage.

4.2. Memory Footprint

In REMORA, we have made a great effort to maintain memory costs as low as possible. The first step of this effort is to avoid creating meta-data structures, which are not beneficial in a static deployment. Distinguishing unicast events and multicast events has also led to a significant reduction in memory footprints as REMORA does not need to create any supporting data structure for unicast events.

The memory footprints in REMORA is categorized into a minimum overhead and a dynamic overhead. The former is paid once and for all, regardless of the amount of memory is needed for the application components, while the latter depends on the size of application. Table 3 shows the minimum memory requirements of REMORA, which turn out to be quite reasonable with respect to both code and data memory. As mentioned before, our sensor node, TelosB, is equipped with $48KB$ of program memory and $10KB$ of data memory. As Contiki consumes roughly $24KB$ (without µIP support) of both these memories, REMORA has a very low memory overhead considering the provided facilities and the remaining space in the memory.

Table 4 shows the memory requirement of different types of modules in the REMORA framework. The exact memory overhead of REMORA depends on how an application is configured, *e.g.*, an application, containing one single instance event producer and one unicast event, needs extra 56 bytes (38 + 8 + 10) of both data and code memory. Ordinary components do not impose any memory overhead as REMORA does not create any meta data structures for them. For other types of modules, REMORA keeps the data

TABLE 4. The memory requirement of different entities in REMORA.

Entity		Code Memory (bytes)	Data Memory (bytes)
Ordinary Component		0	0
Event Producer	Single Ins.	38	8
	Multiple Ins.	42	10
Event	Unicast	0	10
	Multicast	0	10
Multicast Event Consumer		30	6
OS Event		28	4
System API		4	0

memory overheads very low as this memory in our platform is really scarce. We also believe that the code memory overhead is not significant since a typical WSN application is small in size and it may contain up to a few tens of components, including ordinary components. It should be noted that componentization itself reduces the memory usage by maximizing the reusability degree of system functionalities like the one discussed in the code propagation application.

4.3. CPU Usage

As energy cost of REMORA core is limited to only the use of the processing unit, we focus on the processing cost of our approach and show that REMORA keeps the CPU usage at a reasonable level, and in some configurations it even reduces CPU usage compared to the Contiki-based application development.

To perform the evaluation, we set up a Blink application in which a varying number of mirror components (1 to 15) switch LEDs on and off every second. The two implementations of this application, Contiki-based and REMORA-based, were compared according to a CPU measurement metric. The metric was to measure the amount of time required by one REMORA component and one Contiki process to switch LEDs six times: three times on and three times off. With the less number of switches, we cannot extract the exact timing differences as our hardware platform provides a timing accuracy of the order of one millisecond.

We started our evaluation by deploying an application similar to the one presented in Section 2.1 and measuring the CPU usage based on our metric. In each next evaluation step, we added a mirror Blink component to the application and measured again the time. This experiment was continued for 15 times. We made the same measurement for a Contiki-based Blink application and added a new Contiki Blink process in each step. Figure 18 shows the evaluation result of our scenario. When we have one Blink component/process, the CPU overhead of both approaches is almost the same, indicating that the REMORA runtime and framework impose no additional processing overhead. When the number of components/process increases towards 15, reduction in CPU usage is achieved in two dimensions.

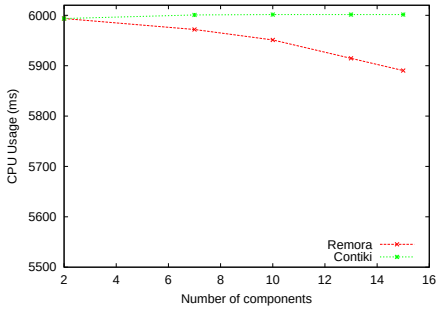

FIGURE 18. The REMORA-based implementation does not impose additional CPU overhead compared to the Contiki-based implementation.

Firstly, the number of CPU cycles for REMORA is slightly less than for the Contiki application. This difference reaches 13 milliseconds when Contiki undertakes running 15 Blink processes. Therefore, we can conclude that REMORA does not impose additional processing overhead affecting the performance of the system. Secondly, the CPU usage of REMORA application is reduced when the number of Blink components is increased. This improvement is achieved because the number of context switches between the REMORA runtime and the REMORA framework is significantly decreased when there are more event producer components (Timer) in PQ.

To clarify this issue, we assume that the application running time is T and Contiki periodically allocates CPU to the REMORA runtime in this period. In each allocation round, the runtime module invokes the event manager in the REMORA framework to poll the application level event producers. Given that there are K producers in PQ, the polling process consumes $K \times t_1$ of CPU, where t_1 is the average processing cost of one element. Therefore, the frequency of event manager calling (equal to the number of context-switches) is in the order of $T/K \times t_1$. Therefore, as the value of K is increased the number of context-switches is decreased accordingly. Figure 19 shows the changes in the number of context-switches when the number of Timer components is increased to 15. As a result, the maximum performance in REMORA relies on the average number of event producer components enqueued during the application lifespan, while in the worst case (a very few producers in the queue) REMORA does not impose any additional processing cost.

5. EXISTING APPROACHES

In this section, we survey the existing component-based approaches for programming on embedded system and WSNs. As mentioned before, a number of these component models are proposed not only to facilitate development of application modules, but

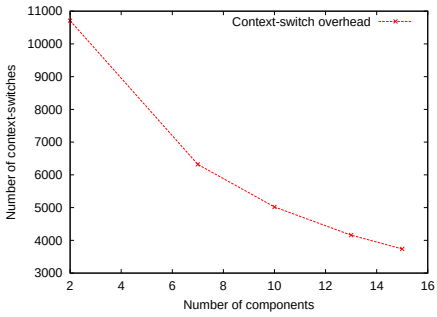

FIGURE 19. As the number of producer components in the queue is increased, the number of context switches is significantly decreased.

also to build component-based operating systems for WSNs. Furthermore, the other objective behind component-based frameworks for WSNs has been the provision of run-time reconfigurability in dynamic WSN applications. There are also a few attempts devoted to porting the existing component-based approaches to other platforms—*e.g.*, embedded systems, large-scale systems, to sensor platforms with some minor changes.

NESC [8] is perhaps the best known component model being designed specifically for WSNs and used to develop TINYOS [18]. Knowing NESC language, programming in TINYOS is quite simple and the developed components are reusable in different applications. As mentioned earlier, the main downside of NESC is that it is tightly bound to the TINYOS platform. Moreover, although NESC efficiently supports event-driven programming, events in NESC are not considered as independent entities with their own attributes and specifications. Therefore, the binding model of event-related components is not well-described as it is not essentially described based on the specification of events. Additionally, the unique features of REMORA, such as multiplicity in component instance and property-based reconfiguration of components bring significant improvements to component-based programming in WSNs compared to NESC.

Coulson et al. in [9] propose OPENCOM as a generic component-based programming model for building system applications without dependency on any target-specific platform environment. The authors express that they have tried to build OPENCOM with negligible overhead for supporting features specific to a development area, however it is a generic model and basically developed for platforms without resource constraints and tends to be complex for embedded systems.

To evaluate OPENCOM, we deployed a sample *beacon* application [19], including Radio, Timer and Beacon components, on a TelosB node with Contiki. Based on our measurements, the memory footprint of this application is significantly high, so that it consumes 4,618 bytes of code memory and 28 bytes of data memory. As a real application, GRIDKIT [20] is an OPENCOM-based middleware for sensor networks, realizing co-ordinated distributed reconfigurations based on policies and context information provided by a context engine. This middleware was deployed on GUMSTIX-based [21] sensor platforms (a resource-rich node type) for a flood-monitoring scenario, where the minimum memory requirement of GRIDKIT core middleware and OPENCOM run-time is about 104 KB of memory. LORIEN [22] is an OPENCOM-driven approach that was recently proposed to provide a fully reconfigurable OS platform in WSNs, however this work is still at an initial stage of development.

TABLE 5. Overview of existing component-based approaches to WSN programming.

Approach	OS Platform	Core Size(KB)	Cost per Component (Bytes)
LORIEN	LORIEN	5.5	350
THINK	OS-Indep.	2	102
FIGARO	CONTIKI	2	15
LOOCI	SUNSPOT	20	587
Remora	**OS-Indep.**	**0.5**	**8**

FIGARO [23] is a WSN-specific dynamic component model, focusing on *what* and *where* should be reconfigured. Specifically, Figaro proposes a set of C macros representing a new component model exploitable over any operating system written in the C language. However, the dynamic aspect of FIGARO—its main feature—is only exploitable on the Contiki operating system. Apart from that, FIGARO fails to consider event management issues at the component design level and mostly relies on the operating system's event handling features.

LOOCI [24] is a component-based approach, providing a loosely-coupled component infrastructure focusing on an event-based binding model for WSNs, while the Java-based implementation of LOOCI limits its usage to the SUNSPOT sensor node.

The THINK framework [10] is an implementation of the FRACTAL [25] component model applied to operating systems. The choice of the THINK framework is motivated by the fact that it allows fine-grained reconfiguration of components. Although the experiments on deploying THINK components on WSNs have been quite promising in terms of memory usage [26], the lack of application-level event support is the main hurdle for using THINK in WSNs.

Table 5 shows a summarized comparison of REMORA with other works proposed in this category in terms of minimum memory required for the core and additional memory overhead per component.

The OSGi model [27] is a framework targeting powerful embedded devices, such as mobile phones and network gateways along with enterprise computers.

OSGi features a secure execution environment, support for runtime reconfiguration, lifecycle management, and various system services. While OSGi is suitable for powerful embedded devices, the smallest implementation, Concierge [28] consumes more than $80KB$ of memory, making it inappropriate for resource-constrained platforms.

OSKIT [29] is a set of off-the-shelf components for building operating systems. OSKIT is developed with a programming anguage called KNIT [30]. However, in contrast to NESC, KNIT is not limited to OSKIT. Nevertheless, OSKIT has adapted the Microsoft COM model and is not primarily focused on embedded systems.

6. DISCUSSION: EXTENSION OPPORTUNITIES

We believe that the current specification of REMORA along with its low resource requirements can tackle the concerns we mentioned at the beginning of this paper. However, there are a number issues—to further support advanced programming in WSNs—that has not been considered by the current REMORA yet. In this section, we focus on these issues and identify potential solutions.

Dynamic Reprogramming. Enabling dynamic reprogramming in WSNs becomes a vital feature when the target application is subject to changes—*e.g.*, fixing bugs, upgrading operating system and applications, and adapting applications behavior according to the physical environment [31, 32, 17]. Although the component-based nature of REMORA can simplify the support for dynamic replacement of system modules, the restrictions on the REMORA component model, including the lack of dynamic memory allocation and the absence of a meta-data to dynamically handle the interactions between components, make the reconfiguration of REMORA components a challenging issue. In fact, the main problem is that how to efficiently provide such a feature in such a way that the overhead of dynamic memory allocation is carefully minimized. Reducing the additional memory required to store the meta-data is another issue in the way of upgrading REMORA to a dynamically reconfigurable module.

Componentization of an OS using Remora. As mentioned earlier, the current goal of REMORA is to be exploited only in application-level programming. However, we believe that the efficient support of event processing in REMORA potentially enables it to componentize system level functionalities. This can also increase the customization of an operating system for a particular WSN application. In the Blink application, we implicitly demonstrated this capability by wrapping the Timer component, which is essentially developed at the OS level. To address precisely this issue, we need to enhance the current REMORA implementation with features like *concurrency support*, *task scheduling*, and *interrupts handling*.

Supporting Preemption. In our current implementation, a REMORA process cannot be preempted by any other process in the operating system. This issue becomes critical when a component execution takes a long time to complete and it causes large average waiting times for other processes waiting for the processor. The event handling model of REMORA can be used to provide preemption by defining a new event type per preemption-required point of application, while in this case the component implementation and the event management become quite complicated. This concern will also be considered in the future extensions for REMORA. In particular, we intend to promote the native Contiki macros, handling process lifecycle, to the REMORA application level. In this way, the REMORA component becomes preemptable by explicitly yielding the running process.

Distribution Support. Beside the fact that REMORA provides a strong abstraction for single node programming, the same level of programming abstraction is expected to occur at the network level. This challenge opens up another key area for future work: how to make REMORA components distributed by the provision of a well-defined remote invocation mechanism. In particular, this refers to rather programming with low-level APIs to provide distribution; we can automatically generate the code which is required for sending data over the network or invoking methods. As a result, the communication strategy could be reified at the architecture level and therefore relieve the programmer from dealing with the specificities of the protocol she/he will need to use for exposing her/his services across the network.

7. CONCLUSION

From a high-level programming point of view, WSNs are still difficult to program. Most of the state-of-the-art programming approaches address this issue by slightly extending low-level system programming languages and promoting them as a solution for application development in WSNs. In this article, we considered WSN high-level programming as a challenge independent from low-level programming paradigms and presented REMORA as a novel programming abstraction for resource-constrained embedded systems.

REMORA simplifies high-level event-driven programming in WSNs by a component-based approach portable to different operating system platforms. Involving PC-based developers in WSN programming and conforming REMORA to the state-of-the-art technologies for component development are two other challenges addressed in this article. The special consideration paid to the event abstraction in REMORA makes it a practical and efficient approach for WSN applications development. The other key features of REMORA include: simplifying middleware services de-

velopment, enabling tunability of operating system software by wrapper components, rich support of component reusability and instantiation, and reduced effort and resource usage in WSN programming.

Careful restrictions on the REMORA component model, including the lack of dynamic memory allocation and avoiding M-to-N communications between event producers and event consumers bring significant improvements to the static deployments in WSNs, where the main improvement happens in sensor memory usage. The main additional memory overhead is induced by the REMORA runtime, occupying only 1% of the total code memory on our sensor platform, which is a very low overhead considering the provided facilities and the remaining space in the memory.

The remora future work targets all issues discussed in the previous section. In particular, we are currently considering the first issue and investigating how the REMORA specification should be modified to support dynamic programming in WSNs with a reasonable cost.

ACKNOWLEDGEMENTS

This work was partly funded by the Research Council of Norway through the project SWISNET, grant number 176151.

REFERENCES

[1] Sugihara, R. and Gupta, R. K. (2008) Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks (TOSN)*, **4**, 1–29.

[2] Szyperski, C. (2002) *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley, Boston, MA, USA.

[3] Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., and Wallnau, K. (2000) Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008. Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, USA.

[4] Van Ommering, R., Van der Linden, F., Kramer, J., and Magee, J. (2000) The Koala Component Model for Consumer Electronics Software. *Computer*, **33**, 78–85.

[5] Genssler, T., Christoph, A., Winter, M., Nierstrasz, O., Ducasse, S., Wuyts, R., Arévalo, G., Schönhage, B., Müller, P. O., and Stich, C. (2002) Components for embedded software: the PECOS approach. *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Grenoble, France, pp. 19–26. ACM.

[6] Hansson, H., Akerholm, M., Crnkovic, I., and Torngren, M. (2004) Saveccm - a component model for safety-critical real-time systems. *Proceedings of the 30th EUROMICRO Conference*, Washington, DC, USA, pp. 627–635. IEEE Computer Society.

[7] Plšek, A., Loiret, F., Merle, P., and Seinturier, L. (2008) A component framework for java-based real-time embedded systems. *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, Leuven, Belgium, pp. 124–143. Springer-Verlag.

[8] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003) The nesC language: A holistic approach to networked embedded systems. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI)*, San Diego, California, USA, pp. 1–11. ACM.

[9] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. (2008) A generic component model for building systems software. *ACM Transactions on Computer Systems (TOCS)*, **26**, 1–42.

[10] Fassino, J.-P., Stefani, J.-B., Lawall, J. L., and Muller, G. (2002) Think: A Software Framework for Component-based Operating System Kernels. *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference (ATEC)*, Berkeley, CA, USA, pp. 73–86. USENIX Association.

[11] Taherkordi, A., Loiret, F., Abdolrazaghi, A., Rouvoy, R., Trung, Q. L., and Eliassen, F. (2010) Programming Sensor Networks Using REMORA Component Model. *DCOSS'10: Proceedings of the 6th International Conference on Distributed Computing in Sensor Systems*, Santa Barbara, CA, USA, pp. 45–62. Springer.

[12] OSOA. The service component architecture. `http://www.oasis-opencsa.org/sca`.

[13] Dunkels, A., Gronvall, B., and Voigt, T. (2004) Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN)*, Tampa, Florida, USA, pp. 455–462. IEEE Computer Society.

[14] Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. (2006) Protothreads: simplifying event-driven programming of memory-constrained embedded systems. *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys)*, Boulder, Colorado, USA, pp. 29–42. ACM.

[15] University of Oslo (2010). The REMORA Component Model. `http://folk.uio.no/amirhost/remora`.

[16] ANTLR. Parser Generator. `http://www.antlr.org`.

[17] Pásztor, B., Mottola, L., Mascolo, C., Picco, G. P., Ellwood, S. A., and Macdonald, D. W. (2010) Selective Reprogramming of Mobile Sensor Networks through Social Community Detection. *Proceedings of 7th European Conference on the Wireless Sensor Networks (EWSN)*, Coimbra, Portugal, pp. 178–193. Springer-Verlag.

[18] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. TinyOS: An operating system for sensor networks. *Ambient Intelligence*, pp. 15–148.

[19] Chatzigiannakis, I., Fischer, S., Koninis, C., Mylonas, G., and Pfisterer, D. (2009) WISEBED: an Open Large-Scale Wireless Sensor Network Testbed. *1st International Conference on Sensor Networks Applications, Experimentation and Logistics (SENSAPPEAL)*, Athens, Greece Lecture Notes of the Institute for Computer Sciences, Social-Inf, pp. 68–87. Springer-Verlag.

[20] Grace, P., Coulson, G., Blair, G., Porter, B., and Hughes, D. (2006) Dynamic reconfiguration in sensor middleware. *Proceedings of the international workshop on Middleware for sensor networks (MidSens)*, pp. 1–6.

[21] Gumstix. `http://www.gumstix.com`.

[22] Porter, B. and Coulson, G. (2009) Lorien: a pure dynamic component-based operating system for wireless sensor networks. *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, Urbana Champaign, Illinois MidSens '09, pp. 7–12. ACM.

[23] Mottola, L., Picco, G. P., and Sheikh, A. A. (2008) Figaro: fine-grained software reconfiguration for wireless sensor networks. *Proceedings of the 5th European conference on Wireless sensor networks (EWSN)*, Bologna, Italy, pp. 286–304. Springer-Verlag.

[24] Hughes, D., Thoelen, K., Horré, W., Matthys, N., del Cid Garcia, P. J., Michiels, S., Huygens, C., and Joosen, W. (2009) LooCI: A loosely-coupled component infrastructure for networked embedded systems. *Proceedings of the 7th International Conference on Advances in Mobile Computing & Multimedia*, Kuala Lumpur, Malaysia, December, pp. 195–203. ACM.

[25] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006) The FRACTAL component model and its support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, **36**, 1257–1284.

[26] Lobry, O., Navas, J., and Babau, J.-P. (2009) Optimizing Component-Based Embedded Software. *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, Washington, DC, USA, pp. 491–496. IEEE Computer Society.

[27] OSGi Alliance (1999). The OSGi framework. `http://www.osgi.org`.

[28] Rellermeyer, J. S. and Alonso, G. (2007) Concierge: a service platform for resource-constrained devices. *ACM SIGOPS Operating Systems Review*, **41**, 245–258.

[29] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. (1997) The Flux OSKit: a substrate for kernel and language research. *Proceedings of the 16th ACM symposium on Operating systems principles (SOSP)*, Saint Malo, France, pp. 38–51. ACM.

[30] Reid, A., Flatt, M., Stoller, L., Lepreau, J., and Eide, E. (2000) Knit: component composition for systems software. *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, San Diego, California OSDI'00, pp. 24–24. USENIX Association.

[31] Taherkordi, A., Le-Trung, Q., Rouvoy, R., and Eliassen, F. (2009) WISEKIT: A Distributed Middleware to Support Application-Level Adaptation in Sensor Networks. *Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS)*, Lisbon, Portugal, pp. 44–58. Springer-Verlag.

[32] Taherkordi, A., Rouvoy, R., Le-Trung, Q., and Eliassen, F. (2008) A self-adaptive context processing framework for wireless sensor networks. *Proceedings of the 3rd international workshop on Middleware for sensor networks (MidSens)*, Leuven, Belgium, pp. 7–12. ACM.