CORMOS: A Communication-Oriented Runtime System for Sensor Networks

John Yannakopoulos[1] and Angelos Bilas[1]
Institute of Computer Science, Foundation for Research & Technology – Hellas (ICS-FORTH)
P.O. Box 1385, Heraklion, Crete, GR-711-10 GREECE
Email: {`giannak, bilas`}`@ics.forth.gr`

***Abstract*— Recently there has been a lot of activity in building sensor prototypes with processing and communication capabilities. Early efforts in this area focused on building the devices themselves and on understanding network issues. An issue that has not received as much attention is generic runtime system support.**

In this paper, we present *CORMOS*, a communication-oriented runtime system for sensor networks. *CORMOS* is tailored: (i) to provide easy-to-use abstractions and treat communication as a first class citizen rather than an extension, (ii) to be highly modular with unified application and system interfaces, and (iii) to deal with sensor limitations on concurrency and memory.

We describe the design of *CORMOS*, discuss various design alternatives, and provide a prototype implementation on a real system. We present preliminary results for resource requirements of *CORMOS* using a pair of sensor devices. We find that the runtime system and a simple network stack can fit in 5.5 KBytes of program memory, occupying about 130 Bytes of RAM. On the specific devices we use, the system is able to process events at a rate of 2500 events/sec. When communicating over the radio transceiver, *CORMOS* achieves a maximum rate of 20 packets/sec.

I. Introduction

Over the past few years, a great deal of attention from industry and academia has been directed toward building networks of ad-hoc collections of sensors scattered throughout indoor or outdoor environments. While the vision of ubiquitous computing [36] remains a research challenge, there have been several recent advances toward this direction.

Improvements in underlying technologies allow us to build battery-powered miniature devices that integrate processing (CPU and memory), small-range RF communication, and sensor-actuator capabilities. The size of today's devices has been reduced dramatically down to inch-level scales and continues to scale with technology. One-inch devices today combine communication, computation, and sensing and actuation functions at low cost. Several networked sensor hardware prototypes have been built by research and development efforts [1]–[3], [5], [7], [19]–[21] and form a foundation for future architectural innovations in this design space.

Despite the progress in device integration and architectural issues, developing applications for such systems remains a challenging task. For instance, deploying a network of sensors in a monitoring application requires developing runtime support in terms of sensing, processing, and communication. Although application-specific support will always require attention, today's sensor networks to a large extend require a full custom design and implementation of the runtime system. Thus, a key missing technology is runtime system infrastructure providing a clear and concise separation of the application and the underlying platform.

Previous efforts in this direction include TinyOS [20] and Mantis [7] that focus mostly on abstracting the devices and providing basic scheduling and communication support. These systems provide fairly low-level interfaces and APIs and treat communication as an extension of the runtime system. Mate [23] and nesC [14] provide high-level abstractions to application programmers while implementing low-level distributed protocols transparently, but impose significant overheads in system resources.

While these efforts have produced invaluable results, there is still much work to be done at the runtime system level for providing convenient and familiar application programming interfaces, geared functionality toward communication-centric applications, and proper management of the limited physical resources available in these systems.

In this work we design and implement *CORMOS* (*C*ommunication-*O*riented *R*untime syste*M* for sens*O*r

[1]Also with the **Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.**

network*S*), a runtime system that aims at providing a convenient programming abstraction that integrates processing and communication, uses simple and unified internal and external interfaces that make it easy to provide new system or application components, and can support systems with strict resource limitations.

CORMOS is built around the notions of events and handlers organized in modules. Modules can extend either application or system functionality resulting in a flat internal organization of various components. *CORMOS* introduces the notion of event paths for supporting communication within and across devices in a network, in a transparent manner. Event paths form the basic means of communication. Moreover, *CORMOS* uses an EDF (earliest deadline first) scheduler that hides all asynchronous operation from modules simplifying system semantics. Finally, *CORMOS* is amenable to efficient implementations both in terms of memory and CPU cycles. We demonstrate how the design of *CORMOS* meets the networked sensor requirements by evaluating it with a pair of communicating sensor devices.

We find that the core runtime system, including support for the event and path abstractions, the system EDF scheduler, RF communication, and library support for various sensors requires about 5.5 KBytes of program memory and 130 Bytes of RAM. Additional modules can be added without modifications to the system. Extending this base system with routing capabilities requires 1768 Bytes of code. Reliable packet transmission requires 1890 Bytes of code. The system can process simple events at a rate of 2500 events/s. Also, it can receive and send packets at a rate of 20 packets/s, resulting in an effective bandwidth of 3.2 Kbits/s (of the maximum 3.5 Kbits/s). The internal *CORMOS* structure makes it easy to port the system on various sensor nodes. For instance, we are able to port *CORMOS* on Crossbow motes [2], as well as on x86 PCs connected with simple RF modules through their serial ports, by: (i) rewriting the RF driver for the new RF modules, (ii) providing new passive libraries for sensor devices, and (iii) recompiling the system.

The rest of the paper is organized as follows. Section II presents our design. Section III describes the platform we use for a prototype implementation and discuss implementation issues. Section IV presents our experiences with using *CORMOS* on a pair of networked sensors. In Section V we present related work and place our contributions in context. Finally, Section VI concludes the paper, also identifying directions for future work.

II. *CORMOS* DESIGN

In this section we describe the abstractions supported by *CORMOS* as well as its internal structure.

A. System Abstractions

Sensor networks may find application in broadly different domains. Therefore, the trend is to build generic sensors that will then be specialized in software. Writing application code for bare sensors requires extensive expertise and is a time-consuming process. Thus, there is a need for runtime system support that provides convenient, high-level abstractions. The basic abstractions that *CORMOS* uses are: *events*, *handlers*, and *paths*.

Events trigger all local and remote actions. An event is an entity that has internal storage, divided in a *frame* that contains event state (values) and an *array list* that comprises a set of handlers that will process the event. Event structures are supplied as arguments to their processing handler(s). Each handler merely operates on the event frame by accessing only the necessary fields. These fields are prescribed at compile-time when the event structure is specified.

Events are created explicitly by handlers when they need to trigger specific actions. However, deallocation is automatic when each event has been processed by all handlers in its path. When events cross node boundaries, they are deallocated in one node and reallocated in the next. Crossing node boundaries occurs through driver modules (RF, UART). Thus, driver modules collaborate with the core runtime system in managing events. However, application modules need not be aware of such issues.

Each event may carry a small amount of arbitrary data (payload). Applications may add data to or remove data from the payload by using a simple stack interface with two primitives, that is, `push` and `pop`, respectively.

Handlers are essentially functions that perform all processing in each node. Each handler is invoked only as a result of scheduling (locally or remotely) an event. Handlers run only through the system scheduler. Each handler takes as input the event that caused its invocation. During execution, handlers can access module variables, create and schedule new events, or call library functions. Handlers are atomic with respect to other handlers, they run to completion and should never block. Their execution may be interrupted only by interrupt service routines associated with hardware device interrupts. Since the only sharing possible across modules is by means of events, interrupt service routines do not interfere with module handlers.

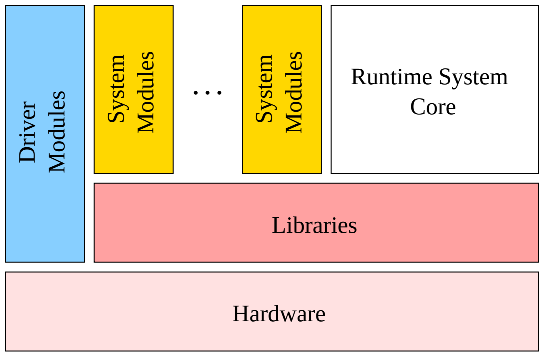

Fig. 1. The structure of a *CORMOS* instance includes the runtime system core, driver, system, and application modules, and libraries.

Given the communication-centric nature of sensor networks, *CORMOS* introduces the concept of *event paths* or (paths) as a first class abstraction. A path represents the flow of data from a source to a destination specifying the modules an event will traverse. Paths allow source modules to specify a sequence of events that will occur in the system and the order in which they will occur. Each path component is a *(module_id, handler_id)* pair. The same pair can be involved in several paths. Thus, the notion of a path is more flexible than forming static module (or handler) stacks or graphs, as is the case in other I/O protocol stacks [15], [17], [22], [27]. An event path is specified at event creation time. The current API provides the ability to also explicitly terminate a path.

B. System Structure

A second issue in sensor networks is that the internal runtime system structure and interfaces are important for extending and tailoring the system. Simpler interfaces allow for easier understanding of system semantics from application module writers. To that end, *CORMOS* uses the event/handler notions to impose a flat internal structure, where all system and application components are represented as modules communicating by events.

CORMOS consists of the following types of components: *modules*, *libraries*, and the *runtime system core*. Fig. 1 depicts this structure and the specific system components currently supported.

Modules encapsulate a particular operation that is either an extension of the runtime system or part of a user application. Each module consists of an *initialize* function, a set of *events* and their associated *handlers*, and a bundle of *static variables* and *functions*. The initialize function is invoked at module start-up and is responsible for registering module's handlers within the module registry. The module variables may be updated by any handler (or function) within the module but are not visible outside the module.

Driver modules are a special category of system modules that use hardware interrupts and their service routines to abstract system devices and hide the low-level hardware interfaces. Driver modules need to be modified when porting *CORMOS* to a new platform. However, this requires limited effort. For instance, we are able to port *CORMOS* to regular x86 PCs connected by simple RF devices through their serial ports within a few hours.

An important driver module that is essentially part of the base system is the RF module. It performs communication operations under the event API by exporting two handlers that other modules can use. The RF driver module maintains a statically allocated buffer for storing the bits of each incoming packet. When a packet is received, it creates a new event and posts it to the scheduler. This design decouples RF communication from application (and system handlers). Thus, there is no need for extensive buffering in the communication layer and explicit receive operations by handlers. This is an advantage of our event-based abstraction and the fact that invoking local and remote events happens in a transparent manner.

Libraries are simple collections of "passive" code that may be used in any part of the system. Currently the *CORMOS* libraries include code for abstracting peripheral devices, such as sensors (light), LEDs, ADC (polling-based), and LCD, that do not require interrupt handlers.

The *runtime system core* consists of the system *scheduler*, *memory allocator*, and *module registry*.

The *scheduler* dispatches events to the appropriate handler, as specified by the event path. Upon exiting, each handler posts the event to the scheduler and the latter identifies the next handler in the path. If the handler is the last path handler (i.e., the path is empty), the event is deallocated. The two main issues behind the design of the scheduler are:

(i) Simple semantics for handler execution: Generally, a scheduler can use either a single or multiple contexts (threads) for scheduling events. Furthermore, in the case of multiple contexts it may or may not be preemptive. Using multiple contexts requires space for maintaining each thread's stack and private state. Furthermore, using preemption requires an interrupt handler to preempt the current thread and switch to the next available. In *CORMOS* we evaluate all three possibilities in terms of memory footprint and context switch overhead, as presented in Section IV. We choose to use a single

context for scheduling all events for reducing memory footprint and simplifying handler semantics.

(ii) Simple interface for scheduling events: A frequent operation in sensors is periodically triggering an event or scheduling an event in the future. One approach to deal with this is to expose to applications the notions of timers and interrupts, as happens in most sensor runtime systems. *CORMOS* hides these low-level abstractions under an *earliest-deadline-first* (EDF) scheduler [24]. Handlers have the option to schedule events with a specific delay by using a simple call that specifies the event and the delay. Periodic scheduling is achieved by rescheduling the event during handler execution (usually at the end of the handler). Since *CORMOS* does not support hard deadlines, an event may execute at a later time than its deadline specifies. This simple mechanism eliminates the need for using lower level, timer-based interfaces in application modules and at the same time requires little resources.

Currently, the *CORMOS* scheduler puts the CPU to sleep when there is no pending computation, but leaves the peripherals operating, so that any of them can wake up the system. Once there is no event to run, a new event can be scheduled only as a result of a hardware device triggered activity that occurs at driver modules. However, this aspect of *CORMOS* requires further investigation.

One issue with the EDF scheduler is to avoid locking when enqueing or dequeing events. As aforementioned, driver modules may use internally interrupt service routines. For instance, this is necessary in the RF driver module, to ensure that bits of a transmitted message are sent at the rate specified by the radio. Moreover, such interrupt routines should be able to also schedule events. Since interrupt routines may run concurrently with module handlers, there is a need to synchronize their accesses to the scheduling queue. One way to do this is to use locks. However, this results in significant overhead in the schedule operation. For this reason, the *CORMOS* scheduler uses two queues, one for handlers and one for interrupt routines. Using two queues is adequate, because interrupt routines cannot be interrupted. The two scheduling queues are lock-free, single pointer queues, sorted by deadline.

The system *memory manager* maintains a static table for allocating and deallocating events. This table's size is specified at compile-time in order to match diverse application needs in the number of events required.

The *module registry* maintains a static table for storing information about all active modules and their handlers. Modules may operate on other modules through the module registry API. A module may load or unload other modules as well as register or unregister handlers during execution. This allows a designer to activate or deactivate a particular routing protocol or reliable protocol on demand as a module.

This flexible structure is especially useful for dynamically reprogramming sensor functionality in future extensions of *CORMOS*. Currently, modules that constitute the user application and core system extensions are loaded during system initialization within the `main` function.

TABLE I
SUMMARY OF THE ATMEL SMARTRF FEATURES.

MCU (90LS8535)	
MIPS	up to 8 (at 8 MHz)
Flash	8 KBytes
RAM	512 Bytes
Current Drawn (at 3V)	6.4 mA active 1.9 mA idle 1 μA inactive
Radio (TRX01)	
Radio Bandwidth	up to 20 Kbps (without Manchester encoding)
Current Drawn (at 3V)	35 mA Tx / 29 mA Rx active 3 μA sleep 0.5 μA inactive
Other Components	
Sensors	photo + temperature (not mounted)
LEDs	red + green + yellow
LCD	16x2
Other	2 push-buttons + 1 encoding wheel

III. IMPLEMENTATION

In this section we present the hardware platform we use for implementing a prototype of *CORMOS* and discuss implementation issues.

A. Hardware Platform

We use two ATMEL SmartRF sensor boards [5]. Fig. 2 shows a schematic diagram of the SmartRF platform and Table I summarizes its features. Each board features a 4 MHz AVR 90LS8535 microcontroller, a TRX01 radio, a light sensor, three LEDs, and a liquid crystal display (LCD).

The processor [4] is highly memory constrained; it contains 8 KBytes of instruction memory and 512 bytes of RAM. It offers three sleep modes: (i) *Idle*, which shuts off the processor. (ii) *Power down*, which shuts off everything but a watchdog timer and the asynchronous interrupt logic necessary for external wakeup. (iii) *Power save*, which is similar to the power down mode, but additional interrupt sources may trigger a wakeup as

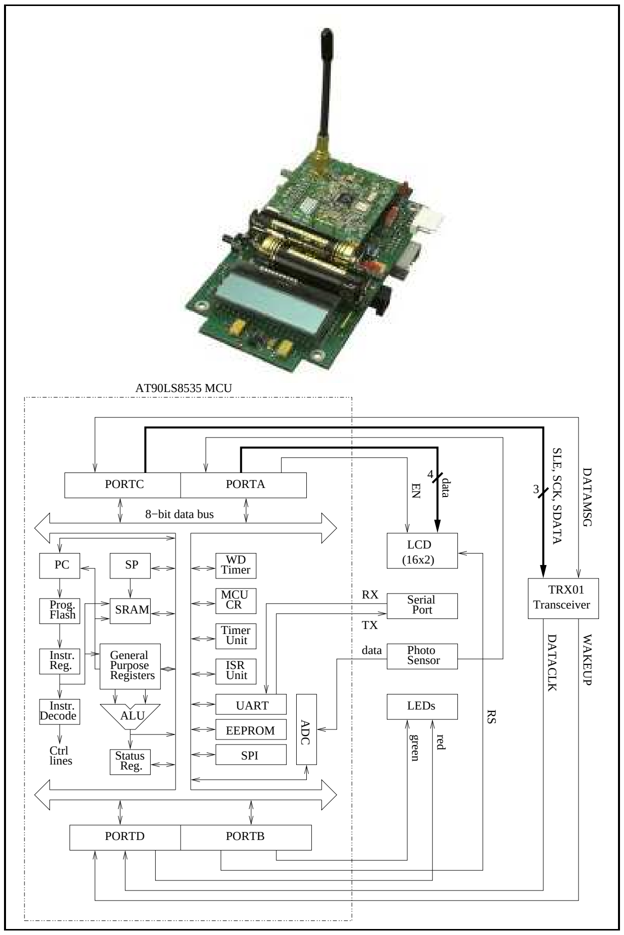

Fig. 2. Photograph and schematic of the ATMEL SmartRF platform.

well. For the last two modes the system requires the presence of an external (off-chip) clock that is not provided by the ATMEL SmartRF boards, and, thus, we currently use only the idle mode.

The RF chip used is ATMEL TRX01 multi-band (from 400 MHz to 950 MHz) transceiver [6]. We use the radio in single channel mode at 915 MHz. TRX01 has three interfaces with the processor: (i) A 3-wire interface (SLE/PortC.1, SCK/PortC.2, and SDA/PortC.3) through which the processor controls the TRX01 registers; (ii) a 2-wire interface (DATAMSG/Port C.0, DATACLK/Port D.2) used to transmit and receive data between the processor and the RF chip; (iii) a 1-wire interface (WAKEUP/PortD.3-external interrupt source) used to wakeup the processor in case a special sequence is detected.

TRX01 features two modes of operation; In *normal* mode, TRX01 functions as a pipe between the microcontroller and the physical link (air). In this mode, it simply passes all bits received in one end of the pipe to the other end of the pipe and no data is stored or processed into the chip. The normal mode requires a DC-balanced (i.e., equal number of ones and zeros) input signal for the receiver to recognize the incoming bits correctly. For this reason, the system uses (software) Manchester encoding, when in normal mode.

TRX01 may also be set up by the microcontroller in a special low-power, *sleep* mode. In this mode, the chip wakes up periodically by means of an internal timer without intervention by the microcontroller. It then checks for a predefined message pattern. If the pattern is not detected, TRX01 goes back to sleep mode. If the correct pattern is detected, an interrupt is generated on the WAKEUP pin that wakes up the microcontroller. In this mode of operation the wakeup message need not be Manchester encoded.

B. RF Driver Module

The RF driver module performs the function of transmitting packets over the radio in a best-effort manner. It exports two handlers that other modules can use to send and receive events. Because handlers in *CORMOS* execute non-preemptively, its long-latency send operation is *non-blocking*: Message transmission is initiated by the `send` handler. Completion is signaled to the module inside from the send interrupt service routine (by calling its local `sendDone` function). Upon reception of a packet, the RF module creates a new event, extracts the event from the packet, and dispatches the event to the scheduler using the event API. Additionally, it performs simple error checking.

The RF driver module can only handle a single message at a time. If a message transmission is in progress, a request to transmit an additional message will not be satisfied immediately. This is done: (i) to avoid buffering and (ii) due to the half-duplex operation of the radio. Instead, the RF module reschedules transmission to happen after a random backoff interval.

In our *CORMOS* prototype, we exploit the *sleep* mode of operation of the radio in order to provide a power-efficient implementation. TRX01 transceivers feature an embedded wakeup protocol, which is based on a header recognition and analysis: A 10-bit sequence (1010100001) is sent before any wakeup. The radio hardware of each device checks periodically if there is channel activity. If the latter is true, the chip becomes active and enters into Rx mode. At the sender side, the header must be repeated for long enough to make sure that a full header will arrive during the wakeup window of the receiver. In our experiments, we set the receiver wakeup period to 10 ms and thus, need to transmit the wakeup header for a period of 24 ms [6]. Since it is possible to wakeup a device and send the data at different rates, we transmit the wakeup header at 10 Kbps to meet the low duty cycle requirement of the embedded wakeup protocol. Afterward, we send the data bits at the rate of 20 Kbps using Manchester encoding.

IV. Evaluation

In this section we present our experiences with using *CORMOS* on a pair of sensor nodes. We evaluate *CORMOS* in three ways: (i) abstractions, (ii) modularity, and (iii) resource requirements.

A. Abstractions

To demonstrate the envisioned usage of the *CORMOS* API, we present the implementation of the LightMonitor module, which is shown in Fig. 3. LightMonitor reads the light intensity value from the light sensor and then transmits the sensor value over the radio. Upon receipt of a new reading from another node, it displays this value to the node's LCD. LightMonitor uses the event path API to specify all required actions. There are three handlers:

- `LIGHT_CYCLE_MSR_SEND_RF` that periodically initiates a light sensor acquisition process and sends the measured value to a remote node,
- `LIGHT_MSR_READ` that reads the luminosity from the light sensor, and
- `LIGHT_MSR_RECV` that is invoked remotely upon arrival of a new sensor reading.

Upon loading, the module initialization function executes. Typically, a module registers in the initialization function all its handlers. Upon exit, a handler needs to reschedule its event by calling `Event_schedule()`. This posts the event to the next handler specified in the path, or if there is no such handler, the event is garbage-collected by the runtime system.

`LIGHT_MSR_READ` and `LIGHT_MSR_RECV` are fairly simple and merely use library calls. `Photo_getData()` is a library call that returns the light intensity from the photo sensor as a 10-bit value (10-bit ADC). `LCD_write()` is a library call that prints its supplied value to the LCD. Additionally, they use the `push()` and `pop()` system calls to associate data with events. `LIGHT_MSR_READ` appends the light sensor value to the payload, whereas `LIGHT_MSR_RECV` reads this value into the local `Light_T` structure.

`LIGHT_CYCLE_MSR_SEND_RF` is the main handler and specifies what will be done and when it will be done. It uses the event path concept (`Path_insertTail()`)

```
#include "cormos.h"
#include "hardware.h"
HANDLERS = {
  #include "LightMonitor.config.h"
};
typedef struct {
  uint16_t val;
} Light_T;
INIT (LightMonitor) (void)
{
  uint8_t i;
  for (i=0;i<HANDLERS_NUM(LightMonitor);i++){
    Event_register(Module_id(LightMonitor),
      HANDLER_ID(LightMonitor, i),
      HANDLER_FUNC(LightMonitor, i));
  }
}
HANDLER (LIGHT_CYCLE_MSR_SEND_RF)(Event_T * e)
{
  Event_T * evnt = Event_alloc ();
  evnt->src  = LOCAL_ADDR;
  evnt->smid = Module_id (LightMonitor);
  evnt->dst  = REMOTE_ADDR;

  Path_insertTail(&evnt->path,
       Module_id(LightMonitor),
       Handler_id(LIGHT_MSR_READ));
  Path_insertTail(&evnt->path,
       Module_id(Route),
       Handler_id(ROUTE_FORWARD));
  Path_insertTail(&evnt->path,
       Module_id(RF),
       Handler_id(RF_SEND));
  Path_insertTail(&evnt->path,
       Module_id(RF),
       Handler_id(RF_RECV));
  Path_insertTail(&evnt->path,
       Module_id(Route),
       Handler_id(ROUTE_RECV));
  Path_insertTail(&evnt->path,
       Module_id(LightMonitor),
       Handler_id(LIGHT_MSR_RECV));
  Path_insertTail(&e->path,
       Module_id(LightMonitor),
       Handler_id(LIGHT_CYCLE_MSR_SEND_RF));
  Event_schedule (evnt, NOW);
  Event_schedule (e, 10000);
}
HANDLER (LIGHT_MSR_READ) (Event_T * e)
{
  Light_T * msg = push (e, Light_T);
  msg->val = Photo_getData ();
  Event_schedule (e, NOW);
}
HANDLER (LIGHT_MSR_RECV) (Event_T * e)
{
  Light_T * msg = pop (e, Light_T);
  LCD_write (2, (uint8_t *)&msg->val);
  Event_schedule (e, NOW);
}
```

Fig. 3. The LightMonitor application example.

to trigger a set of actions: First, get a new reading in the local node (`LIGHT_MSR_READ`), send it to the remote node (`ROUTE_FORWARD, RF_SEND`), and display it on the LCD (`RF_RECV, ROUTE_RECV`, `LIGHT_MSR_RECV`). Finally, it reschedules itself after 10 seconds, which results in periodic execution of this light measurement process.

B. Modularity

The *base* instance of *CORMOS* includes the runtime system core, the RF driver module, and a set of simple libraries. To demonstrate how the system may be extended using the module interface, we develop two additional system modules to extend the basic network stack and provide routing and reliable transmission capabilities. These modules may be combined arbitrarily to match application communication needs.

a) Routing Module: This module uses a SPEED multi-hop routing protocol [16] in its simplest form. It does not support the flow control mechanisms and real-time guarantees that are included in SPEED. Our implementation of SPEED aims at reducing memory requirements. SPEED uses the notion of direction when making routing decisions, and thus, nodes need not maintain per node routing information, as in proactive routing protocols [28], [29]. Rather, each node maintains a local neighborhood table and locating the next forwarding node is a simple lookup operation in this table. In our routing module, routing information is included with every message that is forwarded.

b) Reliable Transmission Module: We have also developed a module that provides reliable transmission. Events are assigned a sequence number. Transmitted events are buffered at the sender until they are acknowledged by a handler that runs periodically at the receiver. The reliable transmission module first checks for unacknowledged events in the send path (retransmit timeout expired) and then sends ACKs for the currently unacknowledged events in the receive path (ack timeout expired). Finally, the receiver does not buffer events received out of order but rather drops them.

C. Resource Requirements

1) Memory Requirements: Table II shows a breakdown of the memory required for each system component. The single-context scheduler implementation, including the EDF policy uses 1244 Bytes of program memory and 48 Bytes of RAM.

In our design we experiment with providing multiple contexts and preemption. As Table III shows, adding multiple contexts requires 902 Bytes of memory for code. With 4 contexts, data size is about 42 Bytes. Adding preemption requires an additional timer handler that occupies 70 Bytes. Performance-wise, a context switch requires mainly saving and restoring all registers (32 general purpose registers plus one control register) resulting in a penalty of 192 cycles on our systems. As mentioned in [20], this is too high a penalty for servicing communication transmission in separate contexts. However, although an approach similar to our current approach of incorporating interrupt service routines in the RF driver can be used to address this problem, the memory requirements are still high for multiple context support. Furthermore, context switching and especially preemption complicate semantics for writing application modules.

CORMOS uses a memory allocator that manages fixed-size blocks, used mainly in event allocation. We also examine a memory manager for variable size blocks. Table IV shows the code and data size for each memory manager. Due to the significantly higher requirements for the variable-size memory manager we eventually use the simpler, fixed-size version.

The routing module requires 1768 Bytes of memory for code and 41 Bytes for data. The route discovery and

TABLE II
CODE AND DATA SIZE BREAKDOWN FOR THE COMPLETE *CORMOS* SYSTEM.

Component	**Code Size (bytes)**	**Data Size (bytes)**
System core		
Path API	416	0
EDF Scheduler	1244	48
Module Registry	566	32
Memory Manager	246	10
Runtime_init	16	0
Driver Modules		
RF	2220	36
Timers	124	4
Libraries		
ADC	80	1
LCD	234	2
LEDS	208	1
Hardware_init	278	0
Other	92	6
Total Base	5632	134
System Modules		
Route	1768	41
Reliable	1890	100
Application Modules		
LightMonitor	432	14

TABLE III

CODE AND DATA SIZE BREAKDOWN FOR DIFFERENT SCHEDULER ALTERNATIVES.

Scheduler	Code Size (bytes)	Data Size (bytes)
Single-context EDF	1244	48
Multiple contexts FIFO (w/o preemption)	902	42
Multiple contexts FIFO (with preemption)	972	42

TABLE IV

CODE AND DATA SIZE BREAKDOWN FOR DIFFERENT MEMORY MANAGER ALTERNATIVES.

Memory Manager	Code Size (bytes)	Data Size (bytes)
Fixed-size blocks	246	10
Variable-size blocks	1178	2

maintenance phases of the protocol occupy 284 Bytes for code, and the module can store routing information for up to 8 neighbors. The reliable transmission module requires 1890 Bytes for code and 100 Bytes for data. This includes caches for supporting concurrent reliable transmission to 4 different sources and reception from 4 different destinations.

2) Performance and Concurrency: First, we examine the cost of basic *CORMOS* operations. Table V shows the cost of managing events and interrupts. Scheduling an event is the most expensive operation with a cost of 313 cycles. This operation involves insertion in an ascending priority queue, based on the event's deadline. The rest of the cost is due to error checking and garbage-collection. Event allocation and deallocation cost 250 and 62 cycles, respectively. They involve mainly the memory manager's alloc/dealloc operations, and the event frame and path initialization during allocation. Adding a handler to an event path costs about 125 cycles and involves mainly adding a 2-byte structure to a cyclic array. The interrupt entry and return procedures cost 32 and 35 cycles respectively. They involve mainly operations that save and restore registers in memory.

Fig. 4 shows the maximum transmission rate of our devices. The maximum transmission rate of 20 Kbits/s is reduced to about 12.18 Kbits/s for 64 Bytes messages, due to the mandatory use of Manchester encoding (*normal* operating mode). If we calculate the maximum transmission rate for the payload only, considering the wakeup header as overhead, then the maximum possible transmission rate is about 6.65 Kbits/s for 64 Bytes messages (*sleep* operating mode). Since *CORMOS* packets have a size of 20 Bytes in the Base configuration, the maximum possible transmission rate that can be achieved in *CORMOS* is about 3.5 Kbits/s.

TABLE V

COST OF BASIC OPERATIONS IN *CORMOS*.

Operation	Cost (cycles)	Time (μs)
Event_alloc	250	68
Event_dealloc	62	17
Event_schedule	313	85
Event_register	125	34
Event_unregister	188	51
Path_insertHead	125	34
Path_insertTail	125	34
Interrupt entry	32	8
Interrupt return	35	9

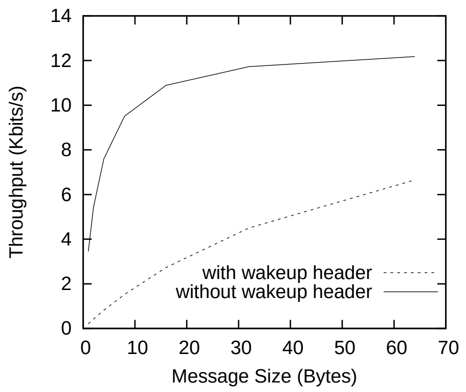

Fig. 4. Throughput for various message sizes with RF module set to *normal* and *sleep* operating mode.

Next we examine the time spent in each system component during the LightMonitor application in two configurations: Base and Base+Routing. We present execution time breakdowns for the application execution for periodic light measurements with periods of 1000 ms, 500 ms, 100 ms, 10 ms, and 0 ms. The experiment lasts 10 sec in each case. Besides the periodic light measurement handlers, we use a *spin* handler which performs busy-wait, i.e., an empty handler that at exit reschedules itself to run immediately, in order to have zero idle CPU time.

Fig. 5 shows how CPU time is divided among components. The system spends 35-40% of the CPU time to the core runtime system. The scheduler dominates this time; it continually posts events and never puts the CPU to sleep due to the existence of the spin handler. 1-20% of the CPU time is used by the transmit interrupt routine of the RF driver. This time increases as

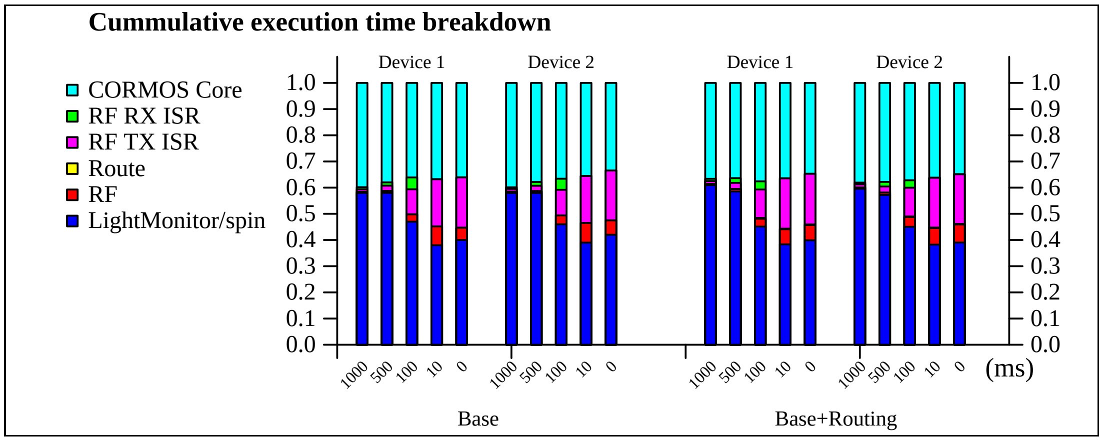

Fig. 5. *CORMOS* execution time breakdown.

the light measurement period decreases, because many more packets are transmitted as the period diminishes. Additionally, each packet transmission is preceded by the wakeup header, further increasing the transmission time. The receive interrupt routine of the RF driver accounts for the 0.5-2.5% of the total CPU time. The RF module's handlers occupy 0.5-5% and the Route module accounts for less than 1% of the total CPU time. The latter happens because we have only a pair of devices so route forwarding is not invoked. Finally, the LightMonitor application module, including the spin handler, runs for the remaining 38-60% of the total CPU time.

Table VI shows the number of events executed and the number of packets sent in the Base configuration. When the light sampling rate is small and thus, there is little network activity, the system is able to process a maximum of about 2500 events/s. The maximum number of packets that can be sent, levels off at about 20 packets/s. Given that we use a payload of 3 Bytes in each packet, this results in a maximum application bandwidth of 480 bits/s. If we take into account the *CORMOS* header as well, which is 17 Bytes, then the effective bandwidth is 3.2 Kbits/s, which is close to the aforementioned maximum achievable RF bandwidth of 3.5 Kbits/s.

TABLE VI
EXECUTION TIME STATISTICS (BASE CONFIGURATION).

Device 1			
Deadline (ms)	**Events**	**Spin Events**	**Packets sent**
1000	25698	25645	10
500	24500	24403	21
100	16010	15532	98
10	12636	11096	184
0	12682	6005	196
Device 2			
Deadline (ms)	**Events**	**Spin Events**	**Packets sent**
1000	25784	25735	11
500	24364	24263	21
100	15945	15466	100
10	12517	10980	183
0	12583	5958	194

V. RELATED WORK

Recently, there has been a lot of research activity in various aspects of sensor networks. We divide this work in three categories: (i) sensor prototypes and components, (ii) runtime system support, (iii) sensor networks and communication, and (iv) other work in neighboring areas.

Currently, there exist numerous prototypes for sensor networks. The authors in [20] build coin-sized sensor node with an ATMEL CPU, RF capabilities, and simple sensors. Smart-Its [10] presents an architecture for sensor nodes and provides various implementations in hardware platforms. Commercial prototypes and products include the ATMEL SmartRF development kit we use in our work [5], the Crossbow motes [2], and the Millennial Net low-power components [1]. Our work is orthogonal to these efforts. Furthermore, we plan to use our runtime system to guide the design of such platforms.

Work at the runtime system level includes TinyOS [20], Mantis [7], nesC [14], Mate [23], and Impala [25].

TinyOS is an event-based runtime system developed to meet the requirements of networked sensors. It is similar to *CORMOS* in that it provides high level abstractions for various system aspects, such as hardware devices, it supports modularity through components, and provides a simple scheduler and a simple communication protocol stack. However, there are also significant differences. TinyOS follows a component model in order to provide software modularity and extensibility. TinyOS uses multiple concepts and abstractions, including commands, events, and tasks. The distinguished roles of various system elements add complexity to the application programming model. Moreover, communication in TinyOS occurs with explicit send/receive operations and an asynchronous timer interface is exposed to applications. In contrast, *CORMOS* uses a unified interface for system and application modules, namely handlers that communicate with events. It also uses the event abstraction to integrate remote communication with processing in a transparent manner. New protocols, such as routing and reliable transmission, may be added to the system as new modules, as can user applications. Finally, unlike TinyOS, *CORMOS* uses an earliest-deadline-first scheduler that makes it easy for applications to schedule events and hides from applications the existence of timer-based, asynchronous interfaces.

Mantis [7] is a Unix-like operating system for sensor networks. It uses Unix-like concepts but is written for resource constrained environments. In contrast, *CORMOS* tries to integrate communication with processing, simplifies the internal system structure using the module abstraction both for system and application components, and is able to support highly memory constrained environments.

nesC [14] is a programming language for sensor networks built on top of TinyOS. It has a C-like flavor and provides support for creating and managing components. Applications are created by wiring components together. nesC performs whole-program compilation and analysis: It compiles an entire application into a single C file, supports aggressive cross-component in-lining, and provides static data-race detection. *CORMOS* is orthogonal to nesC since it assumes the role of the runtime environment.

Mate [23] is a virtual machine for sensor nodes built on top of TinyOS. It uses the code captule concept for integrating communication with processing. Similarly to the *CORMOS* RF driver it uses internal events, hidden from applications, for exchanging messages and hides all asynchrony from applications. In this respect, Mate is close to *CORMOS*, however provides these abstractions on top of the runtime system (TinyOS), whereas in *CORMOS* these abstractions are embedded in the base runtime system. Also, the code captule notion is a higher level concept compared to events in *CORMOS*. Moreover, Mate requires more resources (memory and CPU cycles) for abstracting the underlying sensor node architecture in a virtual machine environment [23].

Impala [25] uses reconfigurable runtime system support for mobile systems, however, targets larger devices that are not as resource restricted as sensor nodes.

There has been significant work in designing and evaluating various aspects of sensor networks and communication protocols. There is considerable work in examining routing in ad-hoc sensor networks [8]. Our goal is not to examine the impact of such schemes, but rather to investigate how runtime systems can facilitate the integration of such schemes as runtime modules. We choose to use one particular routing algorithm, SPEED [16], mainly because we are able to provide a memory-efficient implementation with minimal communication overhead.

Similarly, there has been work in examining the reliability of communication in sensor networks [12], [30], [33]–[35], [37]. In our work we provide a retransmission scheme that guarantees delivery at the data link level. Our goal is not to examine the efficiency of such mechanisms, but rather issues in supporting them in the runtime system. The authors in [18] examine the appropriateness of an Active Messages communication layer for sensor networks. Similarly to *CORMOS* each message results in the invocation of a handler on the receiver side. Unlike *CORMOS*, however, communication happens with a separate (messaging) mechanism, whereas *CORMOS* integrates communication in the event abstraction, allowing modules to schedule events either locally or remotely in a transparent manner.

Other work in the area addresses various aspects of sensor networks. EmStar [13] is an environment that aims at assisting developers in composing applications for sensor networks. Unlike *CORMOS* that assumes the role of the runtime system on sensor nodes, EmStar is a CAD tool that uses existing libraries and runtime system support to reduce development time. MagnetOS [9] is a distributed operating system that aims at building a single system image over a sensor network and provide advanced functions, such as automatic application component placement on sensor nodes. Since MagnetOS is still under development it is not clear what requirements it imposes on system resources. Cougar [11], SINA [31], and Hourglass [32] aim at providing application level support for performing more complex monitoring and querying functions over sensor networks in a uniform and transparent way, i.e. via a sensor query language.

The path concept has been used in the context of networking in x-Kernel [22] and Scout [26], which, however, aims at addressing problems in IP networks. Configurable I/O stacks have been investigated in the context of extensible and stackable filesystems [15], [17], the Click modular router [27], and the x-Kernel [22]. Unlike *CORMOS*, these systems operate at a different domain and under different resource limitations.

VI. Conclusions and Future Work

In this work we present the design and implementation of a runtime system for sensor nodes with processing and RF capabilities. Our system provides easy to use abstractions that integrate communication with event processing, is modular and uses a unified interface for system and application components, and is designed for systems with stringent resource limitations.

A prototype implementation of *CORMOS* fits within 5.5 KBytes of program memory, including the core runtime system with support for our event path abstraction, the EDF scheduler, memory allocator, RF support (without routing and reliable transmission), and simple libraries for hardware devices that do not use interrupts. This base system allows application modules to communicate simply by scheduling events. Our prototype implementation is able to process 2500 events/s and to transfer 20 packets/s on an ATMEL SmartRF platform.

Finally, we believe that although many issues are still open in runtime system support for such systems, two of the main directions that require further work are power-efficient communication protocols and support for dynamic loading and unloading of modules at runtime for adapting the available functionality in each node on demand.

Acknowledgments

We would like to thank Vasilios Garganourakis and Panagiotis Tsakalidis for numerous useful discussions in the course of our work. We would like to acknowledge the support of the General Secretariat for Research and Technology (GSRT), Greece through the ARISTEIA program at ICS-FORTH.

References

[1] Millennial Net, Inc. `http://www.millennial.net/`.
[2] Crossbow Technology, Inc. `http://www.xbow.com/`.
[3] Ember Corporation. `http://www.ember.com/`.
[4] ATMEL 90LS8535 8-Bit RISC processor. `www.atmel.com/dyn/resources/prod_documents/DOC1041.PDF`.
[5] ATMEL SmartRF daughter sensor boards `http://www.atmel.com/products/SmartRF/`.
[6] ATMEL TRX01 Radio Transceiver. `www.atmel.com/dyn/resources/prod_documents/DOC1942.PDF`.
[7] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System Support for MultimodAl NeTworks of In-situ Sensors. In *WSNA*, San Diego, CA, USA, Sept. 2003.
[8] J. N. Al-Karaki and A. E. Kamal. Routing Techniques in Wireless Sensor Networks: A Survey. *IEEE Wireless Communications*, 11(6), Dec. 2004.
[9] R. Barr, J. C. Bicket, D. Dantas, B. Du, T. D. Kim, B. Zhou, and E. G. Sirer. On the Need for System-Level Support for Ad hoc and Sensor Networks. *Operating System Review*, 36(2):1–5, Apr. 2002.
[10] M. Beigl and H. Gellersen. Smart-Its: An Embedded Platform for Smart Objects. In *SOC*, Grenoble, France, May 2003.
[11] P. Bonnet, J. Gehrke, and P. Seshardi. Towards Sensor Database Systems. In *MDM*, pages 3–14, Hong Kong, Jan. 2001.
[12] D. E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A Network-Centric Approach to Embedded Software for Tiny Devices. *Lecture Notes in Computer Science*, 2211, 2001.
[13] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin. EmStar: An Environment for Developing Wireless Embedded Systems Software. Technical Report CENS-TR-9, University of California, Center for Embedded Networked Computing, Los Angeles, CA, USA, Mar. 2003.
[14] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The NesC Language: A Holistic Approach to Networked Embedded Systems. In *SIGPLAN*, San Diego, California, USA, June 2003.
[15] R. G. Guy, J. S. Heidemann, W. Mak, T. W. P. Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *Summer USENIX Conference*, pages 63–71, Anageim, CA, USA, June 1990.
[16] T. He, J. Stankovic, C. Lu, and T. Abdelzaher. SPEED: A Stateless Protocol for Real-Time Communication in Sensor Networks. In *ICDCS*, Providence, RI, USA, May 2003.
[17] J. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, Feb. 1994.
[18] J. Hill, P. Buonadonna, and D. Culler. Active Message Communication for Tiny Networked Sensors. In *INFOCOM*, Anchorage, Alaska, USA, Apr. 2001.
[19] J. Hill and D. Culler. Mica: A Wireless Platform for Deeply Embedded Networks. *IEEE Micro*, 22(6):12–24, Nov.Dec. 2002.
[20] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS*, pages 93–104, Cambridge, MA, USA, Nov. 2000.
[21] J. L. Hill. *System Architecure for Wireless Sensor Networks*.

PhD thesis, UC Berkeley, 2003.

[22] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[23] P. Levis and D. Culler. Mate: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS*, San Jose, CA, USA, Oct. 2002.

[24] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):46–61, Jan. 1973.

[25] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *PPoPP*, San Diego, California, USA, June 2003.

[26] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A Communications-Oriented Operating System. In *HotOS*, May 1995.

[27] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *SOSP*, pages 217–231, 1999.

[28] C. E. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *SIGCOMM*, pages 212–225, Sept. 1994.

[29] C. E. Perkins and E. M. Royer. Ad-Hoc On Demand Distance Vector Routing. In *WMCSA*, Sept. 1999.

[30] Y. Sankarasubramaniam, Özgür B. Akan, and I. F. Akyildiz. ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks. In *MobiHoc*, Annapolis, Maryland, USA, June 2003.

[31] C.-C. Shen, C. Srisathapornphat, and C. Jaikeo. Sensor Information Networking Architecture and Applications. *IEEE Personal Communications*, 8(4):52–59, Aug. 2001.

[32] J. Shneidman and B. Choi. Hourglass: Data Collection Network. `http://www.eecs.harvard.edu/˜syrah/hourglass/sindex.html`.

[33] F. Stann and J. Heidemann. RMST: Reliable Data Transport in Sensor Networks. In *SNPA*, Anchorage, Alaska, USA, May 2003.

[34] K. Sundaresan, V. Anantharaman, H.-Y. Hsieh, and R. Sivakumar. ATP: A Reliable Transport Protocol for Ad-hoc Networks. In *MobiHoc*, Annapolis, Maryland, USA, June 2003.

[35] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks. In *WSNA*, Atlanta, Georgia, USA, Sept. 2002.

[36] M. Weiser. Some Computer Science Problems in Ubiquitous Computing. *Communication of the ACM*, July 1993.

[37] A. Woo and D. Culler. A Transmission Control Scheme for Media Access in Sensor Networks. In *MobiCom*, July 2001.