Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet

Ting Liu Christopher M. Sadler Pei Zhang Margaret Martonosi

Departments of Electrical Engineering and Computer Science
Princeton University
{tliu, csadler, peizhang, mrm}@princeton.edu

ABSTRACT

ZebraNet is a mobile, wireless sensor network in which nodes move throughout an environment working to gather and process information about their surroundings [10]. As in many sensor or wireless systems, nodes have critical resource constraints such as processing speed, memory size, and energy supply; they also face special hardware issues such as sensing device sample time, data storage/access restrictions, and wireless transceiver capabilities. This paper discusses and evaluates ZebraNet's system design decisions in the face of a range of real-world constraints.

Impala—ZebraNet's middleware layer—serves as a lightweight operating system, but also has been designed to encourage application modularity, simplicity, adaptivity, and repairability. Impala is now implemented on ZebraNet hardware nodes, which include a 16-bit microcontroller, a low-power GPS unit, a 900MHz radio, and 4Mbits of non-volatile FLASH memory. This paper discusses Impala's operation scheduling and event handling model, and explains how system constraints and goals led to the interface designs we chose between the application, middleware, and firmware layers. We also describe Impala's network interface which unifies media access control and transport control into an efficient network protocol. With the minimum overhead in communication, buffering, and processing, it supports a range of message models, all inspired by and tailored to ZebraNet's application needs. By discussing design tradeoffs in the context of a real hardware system and a real sensor network application, this paper's design choices and performance measurements offer some concrete experiences with software systems issues for the mobile sensor design space. More generally, we feel that these experiences can guide design choices in a range of related systems.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/Software Interfaces; System Architectures*; C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*Microprocessor/Microcomputer Applications; Real-time and Embedded Systems*; D.4.4 [**Operating Systems**]: Communications Management—*Buffering; Message Sending; Network Communication*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Systems; Real-time Systems and Embedded Systems*

General Terms

Design, Measurement, Performance

Keywords

Sensor Networks, Middleware System, Operation Scheduling, Event Handling, Network Communications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiSYS'04, June 6–9, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-793-1/04/0006 ...$5.00.

1. INTRODUCTION

Energy-aware mobile sensor networks will make a significant impact on society within the next decade. The technology has numerous military and civilian applications that can save lives and improve our overall quality of life. These networks are typically comprised of a low power microcontroller capable of limited information processing, sensors to capture specific data from the environment, memory to store collected data, and a radio to transmit data between nodes. Sensor networks can be comprised of thousands of nodes communicating with each other to relay data and to perform complex distributed calculations. Individual nodes must be extremely energy-efficient as once they are deployed in the environment they are very difficult, if not impossible, to reacquire. Additionally, depending on their size, the environment in which they are deployed, and their power consumption, each node can be extremely unreliable.

Programmers for these systems must make a concerted effort to establish efficient, dependable communications between nodes. Radio transmissions must be minimized as each byte transmitted consumes around two orders of magnitude more energy than each computation on the low power microcontroller. Additionally, protocols must account for the unreliable nature of the network as nodes can fail or simply move out of range during transmission. Steps must be taken to ensure that the network can rapidly adapt to such changes in its structure.

The Princeton ZebraNet Project explores these issues as we develop an energy-efficient mobile sensor network to help track zebra migrations in Africa [10]. Individual nodes have been built into collars and deployed on zebras near the Mpala

Research Centre in Kenya [17]. Each node is equipped with a GPS unit in order to log position information. This information is then passed from zebra to zebra using peer-to-peer protocols until it reaches a base station where it can be processed and analyzed.

Impala—ZebraNet's middleware layer—serves as a lightweight operating system, but also has been designed to encourage application modularity, simplicity, adaptivity, and repairability. A previous study of Impala, prototyped within the HP/Compaq iPAQ pocket PC handhelds, was focused on its middleware support for inherent software modularization, dynamic software adaptation, and remote software update [12]. Impala now is implemented on the ZebraNet hardware featuring energy-efficient components. Critical issues arise concerning implementing software on the real hardware system. Such issues include hardware/software interfaces, system operation scheduling, event handling, and network communication support. This paper discusses these issues in the context of our implementation experiences, and describes how underlying hardware realities impact our design choices. The first test deployment of ZebraNet occurred in January 2004, and the system has evolved to the point where the hardware and software layers have become clearly defined.

Overall, the contributions of this work are as follows:

- We built a system architecture with minimal layering and clean interfaces for resource-constrained mobile sensor systems. The Impala middleware layer transforms rough hardware realities into easy application perspectives. It allows complex hardware controls and accesses to be safely exported as convenient services to applications. It also allows miscellaneous hardware interrupts to be efficiently handled and delivered as a few types of abstract events.

- We established a system activity model that handles a mix of regular operations and irregular events for long-running mobile sensor systems. By having the best knowledge of the overall system activities, Impala uses operation scheduling to achieve the maximum energy conservation without yielding system control to untrusted applications. By processing simple hardware interrupts in short/atomic routines and handling complex software events in long/preemptable routines, Impala achieves both concurrency and prioritization of system activities.

- We developed an efficient network interface that captures and supports the distinctive network communication characteristics of mobile sensor systems. It allows large messages to be transmitted over very small radio packets and supports a range of message models: reliable and unreliable, unicast and multicast, all with the minimum communication, buffering, and processing overhead. By judiciously collapsing traditional network layers, it unifies media access control and transport control to reduce system overhead and achieve performance optimizations.

- We provide solutions for handling the typical resource constraints and hardware realities as are faced by many other mobile sensor systems in software design and implementation.

- Building real working hardware and software has enriched our design decisions and gives us insights we feel have value to the mobile research community as a whole.

The remainder of this paper is structured as follows. Section 2 gives context for the Impala system by presenting the ZebraNet system itself and the system constraints. Section 3 describes the layers and interfaces of Impala. Section 4 discusses the way Impala schedules regular operations and handles irregular events. In Section 5, we present Impala's network interface. Section 6 presents some performance evaluations. Finally, Section 7 presents related work and Section 8 presents our conclusions.

2. THE ZEBRANET SYSTEM

ZebraNet is a mobile, wireless sensor network system aimed at improving tracking technology via energy-efficient tracking nodes and store-and-forward communication techniques [10]. While ZebraNet's most immediate focus is wildlife tracking across large regions with little communications infrastructure, its broader goals concern the deployment, management, and communications issues for large numbers of both static and mobile sensors.

A ZebraNet hardware node includes global positioning system (GPS), a simple microcontroller CPU, a wireless transceiver, and non-volatile storage to hold logged data. ZebraNet does not rely on constant communication access to a base station or other nodes. Instead, it uses periodic node discovery and node-to-node communication to propagate data towards the base station in a store-and-forward manner.

2.1 Hardware Overview

The ZebraNet hardware, which is depicted conceptually in Figure 1 and pictured in Figure 2, is composed of energy-efficient components ideal for use in mobile sensor networks. The major functional components on the board are the microcontroller, GPS, external FLASH, radio, and battery with solar chargers.

To control the hardware, we selected the Texas Instruments Ultra-Low-Power MSP430F149 16-bit microcontroller. This chip has 2KB of RAM, 60KB of internal FLASH memory, and two serial interfaces [24]. It runs off an uninterruptible power supply as we expect it to run continuously.

The microcontroller operates in a dual-clock configuration. It uses an 8MHz clock when accessing sensing, storage, or communication peripherals and a 32KHz clock at all other times. The 32KHz clock consumes half as much power as the 8MHz clock and can be used instead of putting the processor to sleep.

To log the node's position, we selected the μ–blox GPS-MS1E chip for its small size and its ability to quickly acquire locks. It has a typical hot start acquisition time of two to six seconds [4], although our experience has been that good GPS fixes take 10-20 seconds to acquire. The GPS communicates with the microcontroller through an asynchronous serial connection at a rate of 38,400 baud (the maximum rate for this chip) over a port which it shares with the external FLASH. It runs off its own 3.3V power supply which can be turned on and off by the software to conserve power.

To store data, we selected the ATMEL 4Mbit AT45DB-041B DataFlash chip [1]. In our system, the node has enough

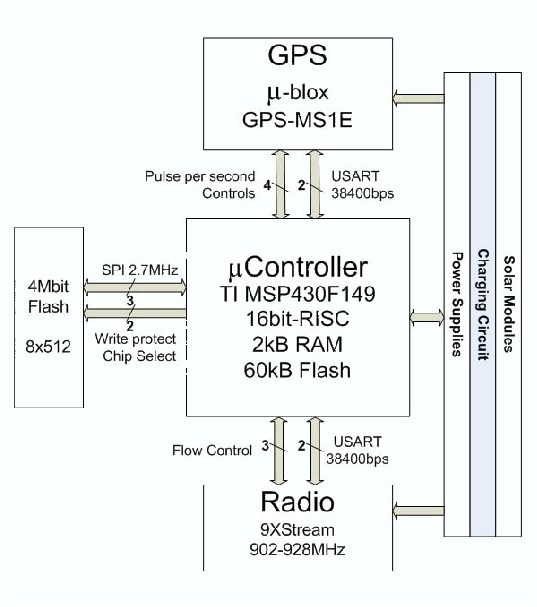

Figure 1: Conceptual diagram of a ZebraNet node.

Figure 2: Photo of a ZebraNet node.

Mode	Current (mA)	Power (mW)
CPU at 32KHz	2.40	9.6
CPU at 8MHz	4.83	19.32
GPS	142	568
Radio transmit	195	780
Radio receive	78.1	312.4

Table 1: Power consumption of hardware components (measured at 4.0V).

memory capacity to store 26 days of its own positional data and 52 collar-days of positional data received from other nodes. The chip communicates with the microcontroller synchronously at a baud rate of 667Kbaud. Sharing the serial port with the GPS allows for the FLASH and the radio to operate simultaneously. The FLASH is powered by the same uninterruptible source as the microcontroller.

To transmit data between nodes, we selected the MaxStream 9XStream radio. It operates at 900MHz and is specified to broadcast data up to 5 miles [15]. In our configuration, however, reliable ranges of 0.5-1 mile are more likely. To transmit, the radio only requires around 1W of power. This is possible because as a receiver, it has a sensitivity of -107dBm. The radio communicates with the microcontroller through the second asynchronous serial connection at a rate of 38,400 baud (chosen to match that of the GPS) and with other nodes at an over-the-air rate of 19,200bps. It runs off its own 5V power supply which, like the supply for the GPS, can be turned on and off by the software to conserve power.

To power the node, we use the Panasonic CGR18650A 2A hour Lithium ion battery [18]. Based on the specifications of this battery, we consider it fully charged at 4.2V and dead at 3.1V. We chose 3.4V as the lower bound of its functional range because at this voltage the radio and GPS units rapidly drain the battery and, consequentially, cannot function properly. The battery is recharged using solar cells strategically placed around the collar.

As energy-efficiency is critical in mobile sensor networks, the ZebraNet hardware features low-power components and efficient power supplies. We measured the power consumption of the system during a cycle in which it performed all possible operations. We applied 4.0V to the board for all measurements and the results are presented in Table 1. Additionally, Figure 3 shows how radio transmissions consume battery power over time.

2.2 Hardware-Imposed Constraints on System Software Programming

The limitations of the hardware system have posed significant constraints on system software programming. Many of them are representative of the challenges in other sensor systems as well.

- **Data and Program memory constraint** The data memory in the microcontroller is only 2KB. This affects the program behavior in many aspects, especially in data buffering. As are used to keep system states and to handle large flows of network data, data buffers often consume large amount of memory, and therefore must be carefully allocated. Additionally, the program memory is only 60KB. This requires software programs to be concise.

- **Energy constraint** The energy budget is tight as we use a solar-array to recharge the battery and to provide the energy essential to achieve the sensing and communication tasks. As is estimated, we are able to fully charge the battery in 50 hours of daylight. This number can vary in either direction, however, depending on the orientation of the solar cells in relation to the sun. Therefore, efforts must be made to maximally save energy, and resorts must be provided to preserve the system when the energy level is severely low.

- **Device access constraint** Device accesses must be carefully scheduled to avoid conflicts which are likely to happen due to hardware limitations. For example, due to voltage regulation challenges, the GPS and the radio should not be turned on at the same time for interference-avoidance purposes. Additionally, the

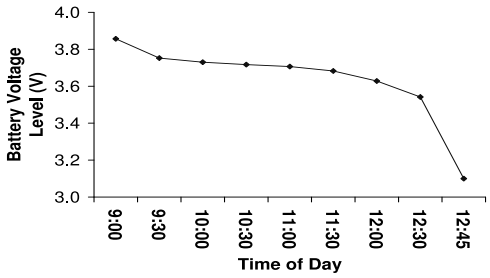

Figure 3: Power consumption of the radio while transmitting constantly for several hours.

GPS and the FLASH share the same serial connection to the microcontroller, and therefore, cannot be accessed simultaneously.

- **Radio packet size constraint** The physical packet size of our radio hardware is only 64 bytes, an order of magnitude smaller than the Ethernet packet size, for example. This means the multi-level packet header in the traditional TCP/IP model will cost a significant communication overhead. Therefore, we need a special network protocol that requires a low overhead to accomplish the essential network communication services.
- **FLASH data storage constraint** For the FLASH memory, new data cannot be written to an address before the data currently at that address is erased, and the smallest erasable unit is a 264-byte page. This means writing data to one location will affect data at other locations. Therefore, a global FLASH organization is required to achieve efficient data storage.
- **GPS sensing time constraint** The time for the GPS unit to acquire an accurate position lock is typically 10 to 20 seconds. This considerable delay in data acquisition implies an asynchronous access and control model is preferred to a synchronous model for operating this sensing device.

3. A STATIC VIEW OF IMPALA: LAYERS AND INTERFACES

The Impala operating system and middleware service model is driven by several issues applicable to ZebraNet and to general sensor network applications as a whole. The first issue is that long-term sensing and communication tasks of sensor network applications require dependable scheduling of regular operations. Sensor networks are designed to run for indefinite periods of time without human intervention. Many sensing and communication operations occur on a predictable timetable. ZebraNet, for example, executes GPS position sensing and wireless radio communication periodically. In addition to these operations, the system must perform many other routine system computations and maintenance. Therefore, Impala must provide clean mechanisms to schedule recurring operations.

A second issue is that sensor network applications require efficient handling of irregular events. Fundamentally, sensor nodes are event-driven systems. Events such as sensor data capture and network data reception occur frequently and are the primary triggers of system computations. An event may result from a single or a sequence of hardware interrupts. Depending on the application programming interface, promiscuous hardware interrupts can be made transparent to applications and delivered only as a few types of abstract events. However, an appropriate event abstraction should balance between simplifying application programming and maintaining the granularity of application-level processing. Additionally, events may be handled by different components of the system, and therefore, require efficient event filtering and dispatching.

Thirdly, sensor network applications require specialized network support. As data gathering is the primary goal of sensor networks, sensor nodes often use aggressive flooding strategies to maximize the chance of finding a path to the desired destination. The resultant multicasts and broadcasts are common communication patterns. Transmissions are required to be reliable in scenarios where data integrity is critical. On the other hand, data can be unreliable in cases where packets can be lost without compromising our goals. For example, peer discovery messages are considered unreliable as the nodes are mobile and may not be in range of the other nodes. Additionally, due to the severe resource constraints and limited hardware capabilities of sensor nodes, efforts must be made to minimize the overhead in communication, buffering, and processing.

The fourth issue is that the complexity of sensor network systems requires dynamic software adaptation. The scale of sensor network systems can be on the order of thousands of nodes; therefore, coordinating the communication and computation across the system is complex. Depending on node topology, network connectivity, and node mobility, over its lifetime the system may encounter a number of different scenarios for each of which a different communication protocol may be appropriate. As such, it is nearly impossible for a single protocol to be appropriate all the time. Some amount of adaptivity is crucial for applications to properly handle an interesting range of possible parameter values.

Finally, the long-term deployment and inaccessibility of sensor network systems require automatic remote software update. It is inevitable that software updates will be required during the lifetime of a sensor network. Because sensors are typically deployed in large numbers in inaccessible places, updates must be deployed wirelessly. Therefore, Impala needs to support automatic remote software updates so that new software can be plugged in at any time. ZebraNet offers very clear motivation for remote software updates, since we clearly do not want to have to tranquilize and re-capture a collection of collared animals each time we need to update the software.

We previously presented Impala focusing on its middleware support for inherent software modularization, dynamic software adaptation, and remote software update, and prototyped it within the HP/Compaq iPAQ pocket PC handhelds [12]. In this paper, we describe Impala as implemented on the real ZebraNet hardware nodes focusing on its operating system functionalities in hardware/software interfacing, system operation scheduling, event handling, and network communication support. Thus far, Impala's dynamic software adaptation and update have only been implemented on iPAQs. We plan to port these to Zebranet nodes in the near future.

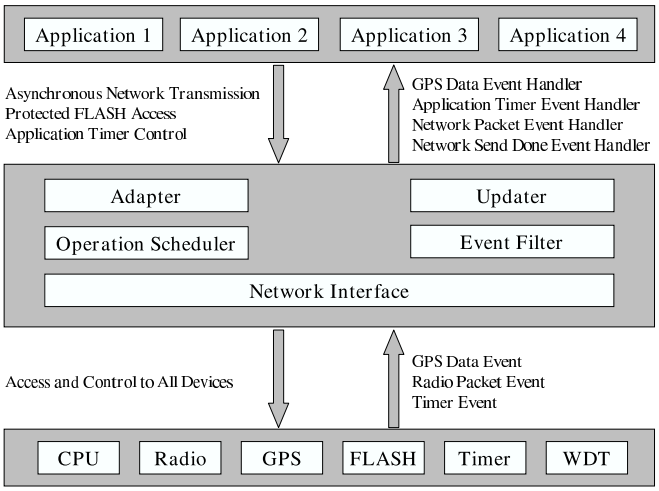

Figure 4: Impala system architecture: layers and interfaces.

3.1 Impala System Layers

Figure 4 shows the static view of Impala with three system layers: the uppermost, application layer, the Impala layer, and the firmware layer. Services and events are the major interfaces between layers. Through the service interface, the firmware layer exports numerous hardware access and control functions to the Impala layer. The Impala layer, however, protects these firmware functions from direct use by the application layer, and only exports the ones needed by applications in a reduced or protected form. It also exports its own network interface to the application layer. The subsections below discuss in more detail the services exported from firmware to Impala and from Impala to applications.

3.2 Services Exported from Firmware to Impala

The firmware layer contains the programs for accessing and controlling individual hardware components. There are six major firmware modules.

The CPU firmware provides Impala with CPU mode control choices based on system performance requirements. The microcontroller CPU in the system can run off an 8MHz clock source or a 32KHz clock source which consumes about half as much power. Impala activates the high-speed clock when the system is performing data sensing and network communication. It switches to the low-speed clock to save power when possible.

The radio firmware provides Impala with the capability to send and receive packets. Data is spooled in and out of the radio in a byte stream. The radio firmware ensures that the byte stream input to the radio is encapsulated correctly into physical packets, and the packets, possibly received from different sources, are restored correctly from the byte stream output from the radio. It signals a packet event to Impala when a packet is completely received.

The GPS firmware provides Impala with an asynchronous interface for obtaining time and position data. First, it configures the GPS unit to start a sensing operation. This operation may take 10 to 60 seconds depending on the time required to get an accurate position fix. Meanwhile, it analyzes the information output from the GPS unit to identify a position fix. If a position fix is obtained, it terminates the sensing operation, saves the data, and signals a GPS data event to Impala.

The FLASH firmware provides Impala with FLASH access and control functions. It partitions the FLASH into five sections for different storage purposes, such as local data, global data, diagnostic information, etc. Data can be sequentially written in and read out of each section in any amount, and can be erased in 264-byte pages or in 8-page blocks.

The timer firmware provides Impala with up to eight software timers. Each timer can be claimed and released by any program. The owner of a timer can set it for an arbitrary amount of time, cancel it, and reset it. Once the timer allocated to the application expires, the timer firmware signals an application timer event to Impala. Timers are the primary mechanism for Impala's regular operation scheduling.

The timer firmware also maintains a system clock, accurate to one millisecond when the CPU is operating on the 8MHz clock, and periodically corrected by the global GPS time. The ability to maintain a globally-synchronized system clock across all ZebraNet sensor nodes allows Impala's network interface to use a simple timeslot-based mechanism for media access.

Finally, the watchdog firmware provides Impala with system monitoring and recovery capabilities. It reboots the system if it is stuck in illegal operations or unexpected fault.

3.3 Services Exported from Impala to Applications

The application layer contains all the applications and programs for ZebraNet. In our first version of ZebraNet, the primary application software running on each node are the communication protocols which log sensor position data and work to propagate the data back to the base station. In future sensor systems, however, application software will be more complex and include precomputation, data filtering and fusing, and database queries in addition to data communication.

Currently, we have implemented a baseline flooding application. For data sensing, it stores the GPS position samples in the FLASH as they become available. For data propagation, it performs periodic, synchronized communication with other nodes. The communication has two stages. In the first stage, each node sends out a peer discovery message through unreliable broadcast. If another node hears this message, that means it has found a neighbor within range to which it can forward data. Therefore, in the next stage, each node floods its position data to all the discovered single-hop neighbors through reliable multicast. To manage data storage, each node uses the local FLASH section for local GPS data and the global FLASH section for GPS data from other nodes. The global FLASH section is used as a medium for our store-and-forward routing scheme. Since new data cannot be written into a FLASH region before the old data is totally erased, and the smallest erasable unit is a 264-byte page, each section is maintained as a cyclic buffer in which data is stored sequentially.

To support these application activities, Impala exports three primary services. First, it exports the system clock and a pre-allocated timer to the application to perform various time-based operations, such as the periodic, synchronized data communication. However, the application is restricted from modifying the system clock or accessing other

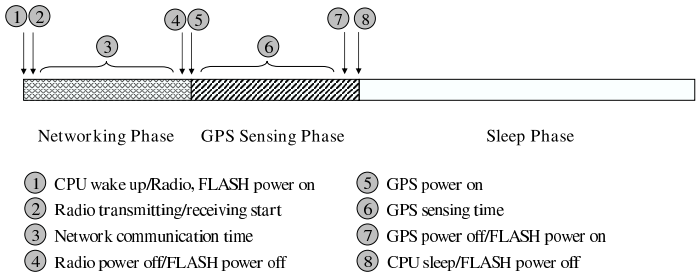

Figure 5: Timeline schedule of Impala regular operations.

system timers as these may interfere with other pre-scheduled system operations. Second, it exports protected FLASH read and write accesses to the application for data storage and retrieval. If the application attempts to access an unauthorized FLASH section or to access the FLASH when it is not available, the access request will be rejected. Finally, it exports an asynchronous network transmission interface that allows the application to pass a number of outgoing messages down to the Impala layer and be notified later when the transmissions are complete.

4. A DYNAMIC VIEW OF IMPALA: OPERATIONS AND EVENTS

The dynamic view of Impala is comprised of regular computing and maintenance operations required by the long-term sensing and communication tasks, and irregular events incurred by the the inherent event-driven attribute of sensor network applications. Thus, Impala's system activity model has two aspects to its personality. For regular operations, Impala acts as an operation scheduler that schedules and coordinates system operations based on application goals, hardware constraints, and energy budget. For irregular events, it acts as an event filter that captures and dispatches events to different system components and initiates chains of processing.

4.1 Regular Operation Scheduling

Impala uses timers to trigger various operations. To ground our discussion in a concrete example, Figure 5 shows a timeline schedule of repeating ZebraNet operations in which a node iterates through a cycle of sending/receiving data, obtaining GPS position, and then sleeping.

When scheduling and coordinating system operations, Impala faces a number of hardware attributes and constraints. First, GPS-aided time calibration allows network-wide operation synchronization. Since ZebraNet sensor nodes have ongoing access to global GPS time, sensor nodes can be easily synchronized. This is especially important for network communication in which all nodes need to turn on and off their radio simultaneously and transmit in assigned timeslots to avoid collisions.

Second, voltage regulation challenges discourage simultaneous radio and GPS operations. ZebraNet's radio and GPS are both high amperage components. Designing a voltage regulator allowing their simultaneous operation is challenging and also leads to extra power loss in the regulator. Therefore, we define a networking phase for radio communications and a GPS sensing phase for GPS sensing to alternate the use of the two devices. We choose to execute radio communications before GPS sensing as the former needs to be synchronized and the latter may take various length of periods.

Third, non-trivial radio wake-up time affects network communication schedule. At the beginning of the networking phase, all sensor nodes wake up the radio from low power mode simultaneously. This takes at least 40 milliseconds [15] and the time can vary on different radios. Therefore, we reserve a fixed period for radio wake-up to accommodate this extra overhead, align the radio transmission and reception across all the nodes, and prevent possible data loss due to incoherent radio states.

Fourth, potentially long GPS sensing time mandates asynchronous GPS sensing operation. In some cases, it takes the GPS as little as 2-6 seconds to get a position fix [4], but typically an accurate fix takes between 10-40 seconds. Because of the large variation, the GPS sensing task is a split transaction. We first perform an asynchronous sensing operation and this is followed some time later by a data delivery event.

Fifth, port sharing of GPS and FLASH prohibits simultaneous GPS and FLASH access. Microcontrollers are commonly pin-constrained, and ours is no exception. In our design, the GPS and the FLASH share the same serial hardware on the microcontroller and an access mode switch is required on that port. For this reason, we must coordinate the operations on these two devices and guard against applications attempting simultaneous accesses.

Finally, stringent energy budget requires energy conservation whenever possible. Energy is always a critical issue in mobile wireless sensor networks. The battery capacity of the ZebraNet nodes can support the full level of system activities for one to three days. The solar array can extend this time indefinitely, but solar cell area is limited, so we need to conserve energy whenever possible. Impala achieves this with two approaches. First, ZebraNet has an 8-minute GPS data sampling interval to ensure that we capture significant movements of zebras while minimizing redundant data records. This determines the desirable frequency and volume of other system activities, and makes it unreasonable in terms of energy consumption for the entire system to be fully active all the time. Therefore, Impala has a sleep phase in which the system is put into low power mode with only minimum resources available for system maintenance. Impala also turns off individual peripherals before they enter a long idle mode. Second, although the energy supply of our system is designed to fulfill the energy consumption under typical conditions, we still need to preserve the system in the case of energy deprivation. Therefore, Impala adapts its operation scheduling to the energy availability. It skips the energy-intensive phases such as networking and GPS sensing if the energy level is inadequate for the subsequent operations.

4.2 Event Handling Model

Impala's event handling model is designed to attack three fundamental issues. First, sensor network systems require an efficient event-based application programming interface. Events are originated from hardware interrupts. Dealing with these interrupts not only involves considerable programming efforts but also requires detailed hardware knowledge. Therefore, Impala has an event abstraction that encapsulates miscellaneous hardware interrupts into abstract

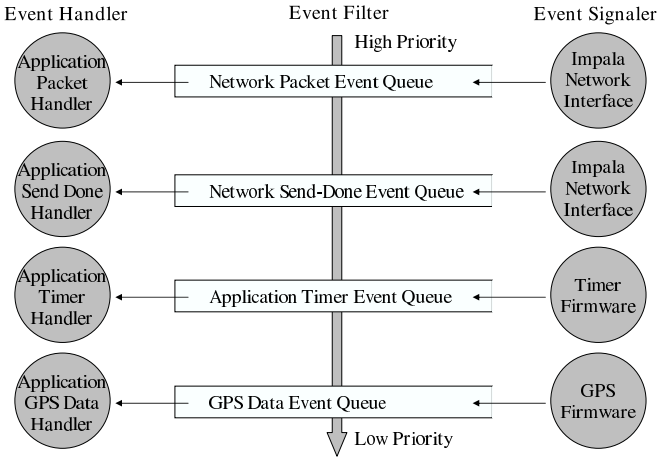

Figure 6: Impala event handling model.

events to simplify application programming while maintaining the granularity of application-level processing.

Impala implements four types of abstract events that are essential for ZebraNet applications. An event is generated and enqueued by an event signaler, dequeued and dispatched by Impala's event filter, and processed by an application event handler. Figure 6 shows the abstract events and Impala's event handling components. A network packet event represents for the arrival of a network packet. Impala's network interface generates this type of events after it receives a packet from the radio firmware and examines the validity of the data. A network send done event represents for the completion or failure of a network message transmission. The network interface generates this type of events after it has completed the transmission or a failure has occurred. An application timer event represents for the time to execute a pre-scheduled application operation. The timer firmware generates this type of events after the application timer expires. A GPS data event represents for the capture of a GPS position fix. The GPS firmware generates this type of events after it analyzes the information output from the GPS unit and identifies a position fix.

Second, concurrency is an inherent attribute of sensor network systems. Information may be simultaneously captured from sensors, manipulated, and streamed onto a network [7]. In addition, some low-level processing has real-time requirements. In ZebraNet, for example, bytes output from the radio will be lost if not processed in time. Therefore, Impala has an hierarchical event handling model that processes simple hardware interrupts in short/atomic routines and handles complex software events in long/preemptable routines. This not only achieves concurrency among multiple flows of processing but also allows low-level processing to interleave with and, if needed, override high-level processing.

For simplicity reasons, our implementation uses a single-thread approach. Hardware interrupt handlers are non-interruptible routines that respond to hardware interrupts and generate software events. Impala's event filter is an interruptible program that runs on the single thread. It constantly checks the incoming events and invokes the application event handlers to process them.

Finally, event prioritization is desirable in sensor network systems. Some events are urgent and require immediate processing, such as the network packet events. Some events are time-constrained, but are not sensitive to small delays, such as the application timer events. Other events are highly latency-tolerant, such as the GPS data events. Therefore, event prioritization allows events with different time constraints to be processed in the desired order. As is shown in Figure 6, Impala's event filter maintains an event queue for each type of events and associates each queue with a priority for event processing.

5. IMPALA NETWORK INTERFACE

The network interface, as a middleware service, is crucial in mobile wireless sensor systems. As in many other mobile wireless sensor networks, ZebraNet uses peer-to-peer communication. Unlike many others, the sparse connectivity caused us to choose pairwise store-and-forward routing, rather than common path-based approaches. To support the application layer which studies various store-and-forward routing strategies, Impala's network interface focuses on the networking model within one hop.

Although networking models have been fully explored in the traditional wireless mobile TCP/IP networks, many distinctive characteristics of sensor network greatly change the design space.

5.1 Impact of Communication Characteristics on Network Sessions

The special communication pattern of sensor network applications like ZebraNet changes the message model. Data gathering is often a major goal of sensor networks. In ZebraNet, in particular, data is frequently collected by the sensing devices, and must all eventually be transmitted to the base station. Sensor nodes often use aggressive flooding strategy to maximize the chance of finding a path to the base station. This leads to the common use of multicast and broadcast protocols. Furthermore, transmissions must be reliable in some cases, but for energy reasons, are preferred to be unreliable in other less critical uses. Impala's network services must support these different uses.

Impala uses session-based transport control. A session is a message designated by the application to have network transaction semantics. Sessions can vary from 1 to 32K bytes, can be unicast, multicast, or broadcast, can transmit data from FLASH or from application RAM buffer, and can use reliable or unreliable transmission. The varying size and styles leverage the MAC layer techniques described in the next subsection. We chose session transmission to be connectionless, because connection-oriented approaches seemed a poor match for the unpredictable motion of sensor nodes in our system. Connectionless sessions also reduces compute and communication overhead which is obviously desirable in sensor systems.

To implement sessions, Impala maintains a send session queue that contains the session descriptors for all the sessions that applications have given Impala. Figure 7 shows the information contained in a session descriptor at the sending node. Each session is assigned with a 4-bit session ID. A session descriptor contains the attributes of a session and, for reliable sessions, keeps track of the networking states of all destinations. The linked list of "destination states" are a set of linked records each holding the per-destination packet/ACK information.

At the receiver side, Impala also maintains a receive ses-

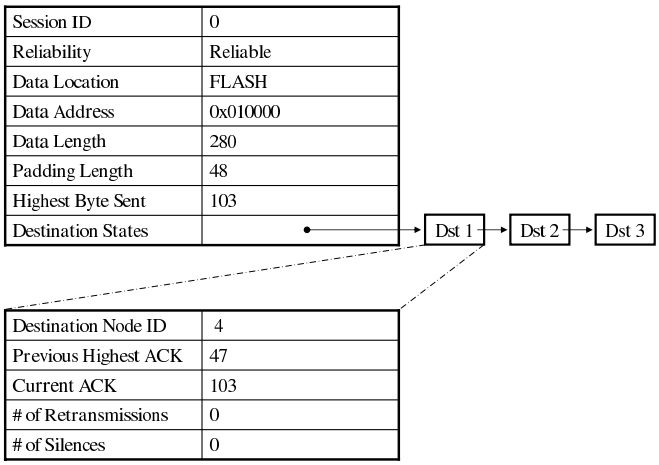

Figure 7: Session descriptor at sending node.

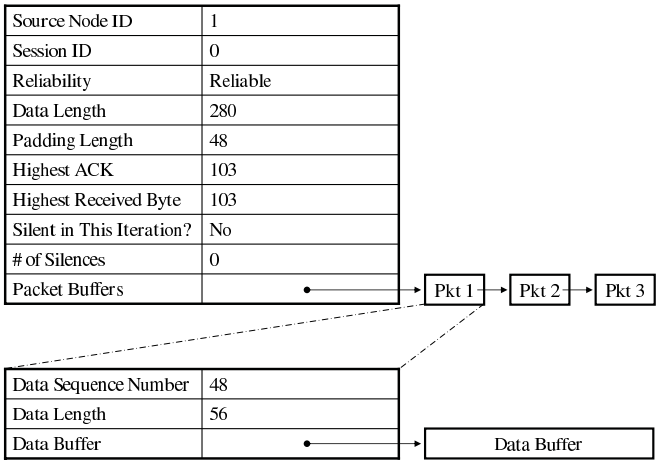

Figure 8: Receiver-side per-session record of packet reception.

sion list that contains the session holders for all the sessions being received from the network. Figure 8 shows the session information stored at the destination node. Each session can be uniquely identified by the source node ID and the session ID. To avoid wraparound, no sensor node can have more than 16 outstanding send sessions. A session holder contains the attributes of a session and also buffers the session packets received but not yet delivered to the application.

5.2 Timeslot-based Media Access Control

Since ZebraNet sensor nodes have ongoing access to globally-synchronized GPS time, sensor node activities can be easily synchronized. Impala takes advantage of this time synchronization and uses simple, round-robin, timeslot-based media access control. We chose this partly for simplicity (code size and energy), but also because the round-robin nature of the approach means that the MAC layer always knows which nodes should be acknowledging reception in each timeslot. This allows simple yet efficient timeout and retransmission mechanisms described in the next subsection.

In the timeslot-based media access control, each node is statically assigned with a unique timeslot for transmission in

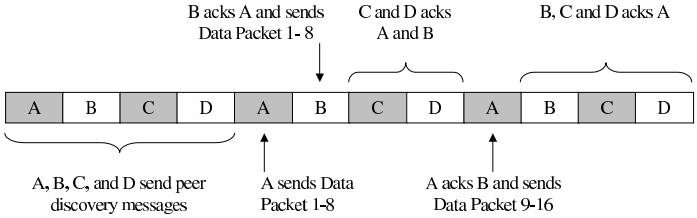

Figure 9: Data sends and related acknowledgments in the timeslot model.

an iteration. A sensor node uses its timeslot to both transmit data packets and acknowledge previously-received packets. Figure 9 shows an example of time-slotted transmissions between multiple sensor nodes. ZebraNet only expects tens of nodes, so this non-scalable solution is acceptable and more efficient. In a larger network, one might choose to use a hybrid time/contention algorithm in which a small number of nodes share a timeslot.

5.3 Optimizing Acknowledgments based on MAC characteristics

Due to processing constraints in sensor systems, as well as the inefficiency of copies in FLASH memory, we choose to compress the traditional layered protocol architecture to reduce data copies and management overhead. In particular, Impala unifies the session-based transport control with the timeslot-based media access control.

Acknowledgment, timeout and retransmission are the mechanisms involved in reliable session transmissions. In each timeslot, the receiver-side Impala scans the sessions and bundles pending acknowledgments into one or more packets. Upon receiving an acknowledgment packet, the sender-side Impala extracts the acknowledgments and updates its per-session ACK records accordingly.

The session-based transport control, which is aware of the round-robin, timeslot-based MAC, adopts a simple but efficient timeout/retransmission mechanism. Since every sensor node has an assigned transmit slot, by the time one gets to transmit, it knows that previously transmitted packets from previous rounds should have already been received and acknowledged by destinations. Therefore, if those packets have not been acknowledged, the node knows that either session or acknowledgment packets have been lost, and retransmits the unacknowledged packets. This timeout and retransmission mechanism allows retransmission to occur at the earliest possible time and improves communication performance. Figure 10 shows the transmission-acknowledgment-retransmission procedure spanning several timeslots between one sender and two receivers.

Timeout mechanisms are important in mobile wireless sensor networks, because nodes may walk away in the middle of a conversation. Impala times out a destination after four retransmissions of the same set of packets or after the destination has been silent through two retransmissions. If all destinations have timed out, the sender-side Impala terminates the session and reports a failure. Likewise, the receiver-side Impala also times out a session after the source has been silent for four timeslot iterations.

5.4 Data Buffering Constraints

Traditional networking models require substantial mem-

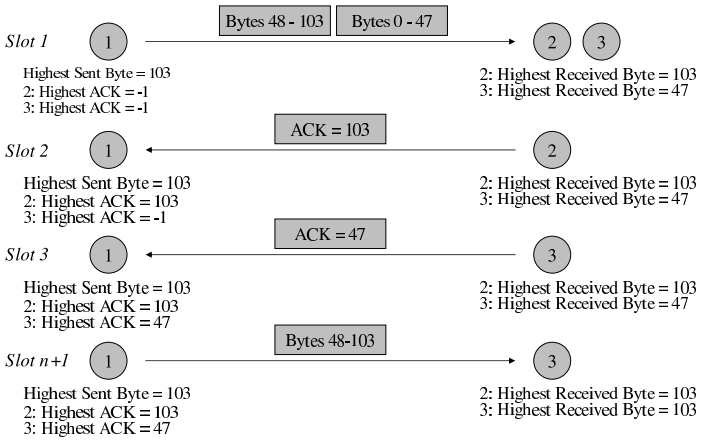

Figure 10: Data send, acknowledgment, and retransmission sequence in ZebraNet.

ory for data buffering. Limited memory in sensor systems necessitates rethinking the data buffering model. For example, ZebraNet sensor nodes have ample FLASH memory, but have only 2KB of RAM. Since FLASH can be delayed, and since there are restrictions on rewriting it, we have adopted two mechanisms to reduce the memory used for data buffering.

The first mechanism we use is to replace data buffering with data indexing. Whether the application data being sent is from the fairly-large FLASH memory, or from the RAM-based application buffer, Impala pins the data, records its memory location, and transmits it directly from there, rather than copying it into a large network buffer. Data indexing minimizes the amount of RAM required for network transmission, and allows the transmission volume, and therefore, the network throughput not to be throttled by the limited memory.

Our second optimization mechanism is to replace session buffering with packet buffering in network reception. Rather than buffering the potentially-huge session before delivering it to the application, Impala only buffers the individual in-order packets. It delivers the packets to the application immediately, even when subsequent packets in the same session are still under transmission. This incremental delivery is also a nice match for the stream-oriented nature of many sensing applications.

5.5 Packets and Packet Event Delivery

Packet delivery, as opposed to session delivery, does affect the application programming style, but we feel it is a good match for the incremental, stream-based processing we anticipate. In traditional TCP-oriented application programming, the application has plenty of TCP buffers and can finish processing one session from one sender before switching to another session from another sender. This programming style is not applicable in ZebraNet, however. Rather than handling session data which should be semantically complete and can be stored in the FLASH memory right away, applications now have to handle packet data which can be incomplete and interleaved by different senders.

In ZebraNet, the small physical packet size of the radio hardware also impacts the design of packet format. Because the natural packets are so small, the traditional multi-level network protocol headers will constitute a considerable communication overhead. We instead adopt a packet format that contains a minimal packet header; this minimal header is essentially a shorthand reference to a previously-sent, full-length packet header that is sent once per session and is applicable to all packets in the session.

Upon receiving a session packet, the receiver-side Impala checks the packet header and tries to associate it with an existing session holder. If there is a match, Impala next checks the packet sequence number; it will buffer an in-order packet and drop an out-of-order or duplicate packet. If this packet doesn't match any existing session holder, Impala will check if the packet contains a session header. If so, it creates a new session holder and buffers the packet. If not, it drops the packet.

5.6 Asynchronous Network Transmission

As mentioned before, Impala has an event-based application programming model in which the application is programmed essentially by creating a comprehensive set of event handlers. All application-level event handlers are required to complete within a limited amount of time for two reasons. First, this allows us to make quality-of-service guarantees about Impala's maximum response time to external events. Second, this helps prevent poorly-behaved applications from livelocking the node. Because the application cannot wait arbitrary periods for lengthy events like network transmissions, we hand these over to Impala and perform them asynchronously. Impala provides applications with an asynchronous transmission model consisting of network hooks for sending data and for learning that a send has completed.

6. IMPALA EVALUATIONS

The Impala middleware system, including the firmware layer, the Impala layer, and a baseline application, has been implemented on the ZebraNet hardware nodes. To evaluate Impala's overhead and performance, we have conducted some preliminary measurements and analysis focusing on the following system aspects: the static and dynamic memory requirements, the operation scheduling and event handling overheads, and the network interface performance.

6.1 Static Memory Footprint

Since our system faces severe memory constraints, it is important for us to minimize code size and RAM usage. Figure 11 shows the program memory and data memory footprints of different system layers.

For program memory, the network interface requires 5712 instruction bytes, and is the largest component in the Impala layer. The FLASH, GPS, and timer modules are the major components in the firmware layer. The application layer is lean because we only implemented one basic application.

For data memory, the network interface in the Impala layer claims 51 bytes of data memory. In the firmware layer, the GPS module requires a 125-byte buffer to receive information from the GPS unit, and the radio module requires a 64-byte buffer to receive packets from the radio.

As depicted in Figure 11, our code currently consumes less than one-third of the total program memory, and we statically allocate less than one-sixth of our RAM which leaves ample memory for dynamic allocation and for future expansions to our system.

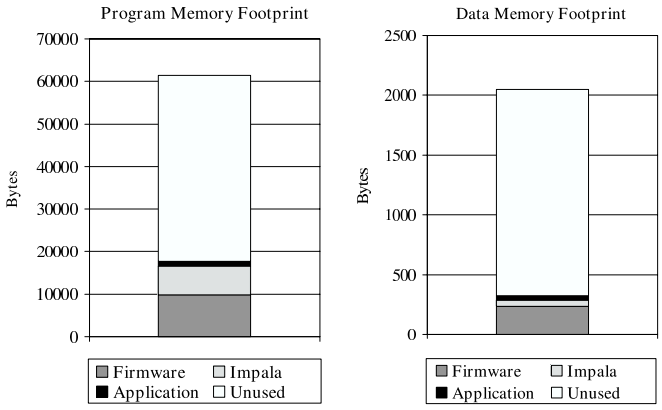

Figure 11: Program memory footprint and data memory footprint.

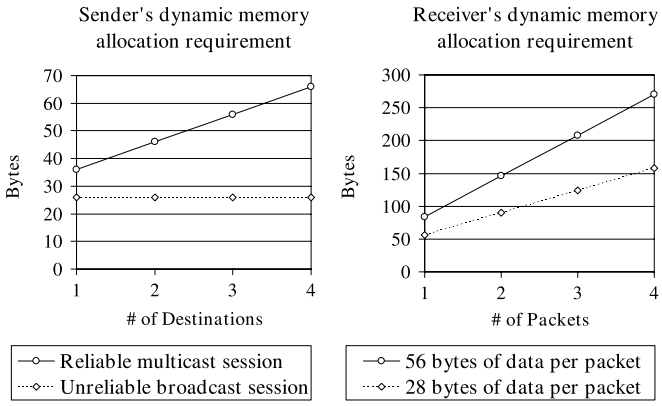

Figure 12: Dynamic memory requirement.

6.2 Dynamic Memory Requirement

Dynamic memory allocation is required by Impala's network interface which claims and releases memory buffers for on-the-fly network transmission and reception.

On the sender side, Impala allocates memory buffers for application sessions passed down for asynchronous transmission. The memory size is linearly related to the total number of destinations for a reliable multicast session, and is constant for an unreliable broadcast session. Figure 12 shows the dynamic memory requirements for buffering sessions with up to four destinations. Since data in the FLASH is indexed rather than buffered by the network interface, the sender side memory requirements are drastically reduced.

On the receiver side, Impala allocates memory buffers for packets that have not been delivered to the application. The memory size is linearly related to the total number of buffered packets. Figure 12 shows the dynamic memory requirements for buffering up to four packets.

We conducted an experiment to examine the queuing status of the received packets, and therefore, the worst-case estimate of the receiver side dynamic memory requirement. In this experiment, we generated a continuous flow of network packets at the highest rate, and monitored the maximum number of buffered packets over time. It turned out that the packet queue contains at most one packet over time. This is because the application-level packet processing including FLASH write and other computations is significantly faster than the over-the-air radio transception rate so that packet delivery is never delayed. Nevertheless, extra packet buffering on occasional basis will still be needed when the system is expanded to perform multiple FLASH or CPU-intensive tasks.

Impala Activity in Event Handling	Time
To deliver and remove a network packet event	88 cycles
To deliver and remove an application timer event	16 cycles
To deliver and remove a GPS data event	17 cycles

Table 3: Event handling overhead.

6.3 Operation Scheduling and Event Handling Overhead

The operation scheduling overhead often comes from manipulating hardware devices before or after a device operation, and setting up timers for a future operation. Minimizing the operation scheduling overhead is essential to the proper implementation of our system. If the scheduling of the operation takes an excessive amount of time, the operation itself may be delayed. This may have unintended consequences such as causing the individual nodes to lose the time synchronization.

Table 2 shows the CPU times for scheduling various Impala operations. The time to turn on the radio for transception is 50ms as Impala has to wait for the radio to wake up from low-power mode. We compensate for this delay by reserving a radio wake-up period at the beginning of all radio communications. The time to power off the radio is 11ms as Impala has to wait for the pin signal that indicates the radio send buffer is empty. All other scheduling overheads are less than 1ms.

We evaluate the event handling overhead by measuring the latency of delivering the application events by Impala's event filter. This overhead determines how fast we can respond to a signaled event. The times to generate and process some of these events are presented in the next subsection.

Table 3 lists the event handling overhead for all application events. Take the network packet event for an example, the handling overhead involves traversal of an event queue, invoking the upper level event handler, and releasing dynamically-allocated memory buffers.

6.4 Network Interface Performance

We evaluate Impala's network interface in its processing overhead for packet reception and transmission, its communication overhead caused by packet headers, and its communication latency in reliable multicast.

6.4.1 Packet Reception Processing Overhead

For packet reception, Impala propagates an incoming network packet from the radio hardware, to the radio firmware, to the network interface through interrupts and callouts. Then the packet is enqueued by the network interface until the event filter dequeues and delivers it to the application. Table 4 shows the processing time to receive a network packet in each system layer.

Understanding the major overhead in each layer requires more detailed analysis. The radio hardware layer processing

Scheduling Type	When	Impala Activity	Time
CPU Scheduling	Beginning of the system active time	To put the CPU on the fast clock	3127 cycles
	Beginning of the system sleep time	To put the CPU on the slow clock	38 cycles
Radio and FLASH Scheduling	Beginning of the networking time	To set up the first transmission time slot, turn on the radio, and wake up the FLASH	50 ms
	Time slots except the last one	To set up the next transmission time slot	260 cycles
	The last time slot	To set up the time for network cleanup and radio and FLASH power-off	265 cycles
	End of the networking time	To clean up unfinished network sessions, power off the radio and the FLASH, and set up the next networking time	11 ms
GPS Scheduling	Beginning of the GPS sensing time	To initiate GPS sensing and set up its finish time	1247 cycles
	End of the GPS sensing time	To format the GPS data, power off the GPS, signal an GPS data event, and set up the next GPS sensing time	2550 cycles

Table 2: Operation scheduling overhead.

System Layer	Time	Processing Breakdown	Time
Application Software	111819 cycles	Packet write to FLASH	110172 cycles
		Non-FLASH related computation	1647 cycles
Network and Event Filter Middleware	1058 cycles	Packet processing by the network interface	970 cycles
		Packet delivery and remove by the event filter	88 cycles
Radio Firmware	3470 cycles	Processing packet synchronization bytes	56 cycles per byte
		Processing intermediate packet bytes	49 cycles per byte
		Processing the last packet byte	369 cycles per byte
Radio Hardware	255585 cycles	N/A	N/A

Table 4: Packet reception processing time by system layers.

Operation Type	Time
`memset()` a 64-byte memory buffer	313 cycles
`malloc()` a 56-byte memory buffer	196 cycles
`memcpy()` 56 bytes between two memory buffers	349 cycles
Read 56 bytes from FLASH	13478 cycles
Write 28 bytes to FLASH	55086 cycles

Table 5: Memory and FLASH-related operation cost.

Packet Type	Time
Unreliable broadcast session packet containing 6 bytes of peer message	1061 cycles
Reliable multicast session packet containing 48 bytes of data padding	596 cycles
Reliable multicast session packet containing 56 bytes of data	970 cycles
ACK packet containing one session acknowledgement	266 cycles

Table 6: Packet reception processing time by the network interface for different packet types.

is basically bottlenecked by the over-the-air baud rate. Our radio transmits and receives at 19.2Kbps, which implies a throughput of 26.67ms per packet. However, the measured packet reception rate is 30.47ms per packet. This means the hardware layer adds 14% extra cost for bit synchronization, checksum computation, etc. The firmware layer receives the packet byte by byte and the overhead mainly comes from processing the last byte as it needs to hand over the entire buffered packet to the upper layer and wipe out the old content of the buffer for the next packet. Table 5 shows the costs of some common memory and FLASH-related operations. The middleware layer is mainly responsible for examining and buffering the packet by the network interface, and retrieving and delivering it by the event filter. Table 6 also shows the cost for examining and buffering different types of packets. As dynamic memory allocation are frequently used by the network interface to buffer new packets, these costs mostly come from memory-related operations. Finally, the application layer's major overhead comes from FLASH access as packets are eventually stored in there.

6.4.2 *Packet Transmission Processing Overhead*

On the transmission side, the network interface provides the applications with an asynchronous operation for network transmission. As we described before, sessions can be dropped into the network interface by the application at any time. The time to drop an unreliable broadcast session, such as a peer discovery message, is 496 cycles. The time to drop a reliable multicast session, such as a GPS data session, is 901 cycles.

When the networking time comes, the network interface will update session states, compute the information to send, copy data between FLASH and RAM, and invoke the radio firmware to transfer data to the radio send buffer. All these operations are in parallel with the actual data transmission by the radio hardware. Figure 13 shows the time spent on each system component for transmitting a packet.

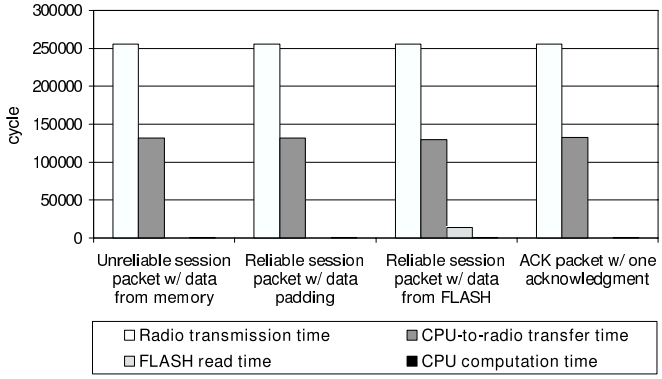

Figure 13: Packet transmission processing time by system components.

Packet Type	Header Size	Payload Size
Session packet	8 bytes	1 - 56 bytes
ACK packet	6 bytes	6 - 54 bytes
Time and voltage packet	6 bytes	6 bytes

Table 7: Header size and payload size for different packet types.

6.4.3 Communication Overhead by Packet Header

Due to the small physical packet size of the radio hardware, packet headers become a significant communication overhead. Impala's network interface uses a special protocol to reduce the header size and improve the data throughput. Table 7 shows the header size and payload size for different packet types. The payload size can vary depending on how much data it carries for a session packet or how many acknowledgments it contains for an ACK packet.

In addition to the packet header, the first packet of an unreliable broadcast session also contains a 4-byte session header, and the first packet of a reliable multicast session also contains eight or more bytes of session header, depending on the number of destinations.

6.4.4 Communication Latency in Reliable Multicast

The network interface is designed to support connectionless reliable multicast which is a common communication pattern in ZebraNet. We conducted a simulation to evaluate the efficiency of our multicast scheme. In this simulation, one node performs reliable multicast to a number of destinations. We compare the communication latency of two multicast approaches. One approach uses a number of reliable node-to-node transmissions, and the other uses Impala's reliable multicast mechanism. Both approaches use the same timeslot-based media access control as in ZebraNet for latency measurements.

Figure 14 shows the observed data delivery latencies for different number of destinations and different packet loss rates. We set the timeslot capacity to be 64 packets and the total transmission volume to be 6400 packets. The packet loss rate is the probability of dropping a packet. It is simulated to account for all packet loss occasions such as network disconnectivity, bad signal reception, bit error, etc. In a baseline scenario when the packet loss rate is zero and there

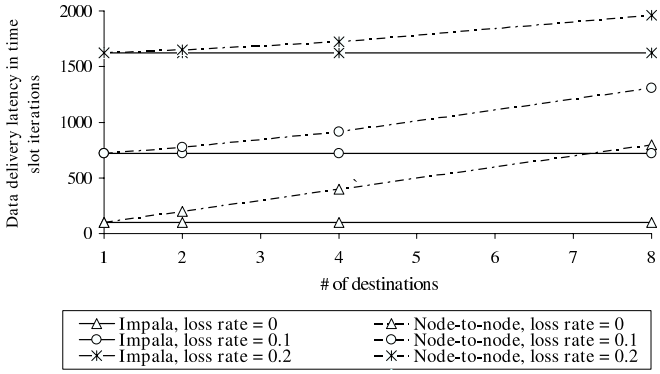

Figure 14: Communication latency in Impala's multicast scheme and node-to-node transmission scheme.

is only one destination, both multicast schemes finish the transmission within 100 timeslot iterations. In other cases, Impala's multicast scheme achieves constant data delivery latency with regard to the number of destinations, while the node-to-node transmission scheme explodes with growing number of destinations. This is because the node-to-node transmission has to split the network bandwidth among different transmission pairs. However, the impact of network bandwidth split becomes less evident as the packet loss rate increases. This is because Impala's multicast scheme is more sensitive to packet loss as it must retransmit to all the destinations once a packet is lost at a single destination.

7. RELATED WORK

We have developed a system that combines aspects from the sensor networking community and from the mobile ad-hoc networking community. In this section, we look at how our system compares to new technologies in both of these distinct fields.

Sensing Hardware: A number of energy-efficient sensor nodes have been developed in the past few years [20][6][16][21]. The two devices that most closely parallel our nodes are Berkeley's Mica2 Mote and UCLA's Medusa-WK2.

The Mica2 Mote has a 4MHz, 8-bit processor and uses the same off-chip FLASH memory chip and serial interfaces as our nodes. Medusa-MK2 features a dual microcontroller scheme which uses the same processor as the Mica2 in situations that require minimal computational power and a 40 MHz ARM processor to operate its on-board GPS unit and other attachable high energy consuming sensors.

However, once deployed, both of the aforementioned nodes are intended to remain close together to form a densely populated network. This allows them to use extremely low-power radios with a very limited range. ZebraNet nodes, on the other hand, are intended to be extremely mobile and distributed over a large area. Having a sparsely populated mobile network demands a more powerful radio with a much larger range.

In addition, due to the high power consumption of our radio and GPS, we cannot hope to run the system continuously for months at a time on one set of batteries. Nor can we reduce the duty cycle of data collection and still achieve our objectives. To compensate, we use a rechargeable battery with solar cells distributed around the collar.

Sensing Operating Systems and Middleware: Various operating systems and middleware layers have emerged to control sensor nodes [5][7][11][13][22]. Two such systems that closely relate to Impala are TinyOS and Maté.

TinyOS, the popular operating system designed to run on the Motes, has many low level characteristics in common with Impala. For example, both Impala and TinyOS place an emphasis on event handling through hardware interrupts and the utilization of on-the-fly processing to conserve memory.

One big difference between our system and TinyOS is a result of the differences in the nodes on which they will be used. The Mote is designed to accommodate a variety of interchangeable sensors. This is reflected in TinyOS's emphasis on concurrency-intensive operations which allow the system to handle multiple flows of data from independent sensors simultaneously. Impala uses a combination of polling and interrupt handling to allow for a similar interleaving of scheduled and unscheduled events. In the ZebraNet system, however, Impala takes advantage of the fact that we have a fixed number of sensors that work in a predetermined fashion by using hardware timers to schedule all major events. This allows us to save a great deal of power through the use of the dual clock scheme and the timely use of energy-hungry components.

Maté is a virtual machine that lies on top of the operating system and is designed to provide a layer of security and a basis for automatically updating nodes via virally propagated programs. ZebraNet does not need the added security Maté provides, but the ability to perform viral software updates was implemented in the original version of Impala [12] designed for palmtop computers and will be implemented on the ZebraNet nodes in the near future.

Protocols and Routing Schemes: Our peer-to-peer routing scheme has roots in a number of proposed routing methods designed to make communication in mobile ad-hoc networks more efficient. DSR [9] and AODV [19] send out route discovery messages which perform similar roles to our peer discovery messages. The difference is that DSR and AODV attempt to discover a complete route to a destination whereas our algorithm only attempts to discover a node's immediate neighbors. This modification is essential to our system because under normal circumstances there will not be a complete route to the base station; rather, data is expected to propagate slowly from node to node until the base station comes into range.

Directed Diffusion [8] uses a data-centric scheme in which messages are passed through the network in a series of independent neighbor-to-neighbor communications very similar to our peer-to-peer transmissions. However, this scheme would not work well in our system because our network has a very low connectivity and our topology is changing too fast for important messages to arrive in a timely fashion.

Sensing Application Studies: In addition to ZebraNet, there are other concurrently running efforts to use sensors to monitor wildlife or in other mobile applications. The VAFalcons project places solar powered satellite transmitters on Falcons and uses satellite telemetry to determine the animal's position [25]. Similarly, the Pacific Ocean Salmon Tracking Project places acoustic tags on juvenile salmon [2]. The sound emitted from the tags is recorded by receivers strategically placed along known migration routes and can be used to closely reconstruct the exact movements of the fish. Both projects, however, rely on an expensive, high-tech fixed infrastructure to gather data; therefore, the nodes do not need to interact with one another.

A stationary sensor network composed of Motes running TinyOS has been deployed on an uninhabited island off the coast of Maine to monitor the nesting habitats of certain birds along with environmental conditions such as temperature and humidity [14][23]. The group conducted a successful multiple month experiment in which data was collected and transmitted through the network using a CSMA MAC layer to protect against collisions. This deployment, along with a similar deployment at the James Reserve in Idyllwild, CA [3], is providing the sensor network community with a great deal of insight into the numerous issues relevant to a real-world deployment.

8. CONCLUSION

This paper presents our implementation of ZebraNet, a system that utilizes mobile sensor networking technologies to monitor zebra migrations on energy-constrained hardware. ZebraNet nodes are controlled by the Impala middleware system, which efficiently handles scheduled operations as well as the event-driven operations characteristic of sensor networks. Through this implementation, we show that this architecture has low overhead and can offer effective improvements on the performance, energy-efficiency, and robustness of the system. In accomplishing this goal, we also show that traditional networking layers can be strategically optimized to help us meet our stringent resource constraints.

Several lessons have been learned from our implementation experiences. First, even with algorithms that emphasize memory efficiency, ZebraNet nodes have the potential to exhaust all available memory. Meanwhile, we have 4Mbits supply of FLASH storage. The technology exists for us to add orders of magnitude more without consuming any more power or area. This added FLASH could then be used in a light-weight virtual memory scheme. A smart algorithm could minimize the number of writes to the FLASH and perform load leveling to extend the life of the FLASH module.

Second, Impala dominates the control on device scheduling and power management. On one hand, this protects the system from being abused by untrusted applications, and yields maximum gain in energy-efficiency as Impala has the best overview of system activities. On the other hand, it may also "hardwire" choices in ways that are awkward to particular applications. Therefore, we plan to design mechanisms for Impala to have input from the applications on the execution of these operations.

Thirdly, in our system, FLASH access and CPU-to-radio transfer are both synchronous and time-consuming operations. Using hardware interrupts to signal the completion of these operations seems a better option in terms of saving the CPU time than having the CPU busy waiting. However, the second one requires a much simpler implementation and yet still does not sacrifice system concurrency or significantly degrade system performance. This is because these long operations are always interruptible so that other computations can break in and exploit CPU cycles when the FLASH access or CPU-to-radio transfer is in progress.

Finally, the reliable multicast mechanism is crucial to our mobile sensor system. However, in sparsely connected mobile sensor networks where packet loss and retransmissions are frequent and network bandwidth is precious, the reli-

able multicast service should be used at the discretion of the application programmers. Depending on data integrity requirements, one might choose to use unreliable broadcast to improve data throughput.

The process of developing real working hardware and software for a highly mobile, sparsely populated network has given us a unique perspective on many issues relating to mobile sensor networking technologies. The design choices and performance measurements presented in this paper offer some concrete experience that can contribute to the existing wealth of knowledge in the sensor networks research community.

ACKNOWLEDGMENTS

This work was supported in part by an NSF Information Technology Research grant (ITR-0205214). In addition, we gratefully acknowledge the Mpala Research Foundation for their support of the Mpala Research Centre which has enabled our deployment of this system.

9. REFERENCES

[1] ATMEL. AT45DB041B, 4M bit, 2.7-Volt Only Serial-Interface Flash with Two 264-Byte SRAM Buffers data sheet. `http://www.atmel.com/`, June 2003.

[2] Census of Marine Life. POST: Pacific Ocean Salmon Tracking Project. `http://www.postcoml.org/`, 2003.

[3] Center for Embedded Networked Sensing. Research infrastructure: James reserve local area power system and network enhancements. `http://www.cens.ucla.edu/Project-Descriptions/Research_Infrastructure/index.html`.

[4] P. Eggenburger. GPS-MS1E Miniature GPS Receiver Module Data sheet. `http://www.u-blox.ch/`, Oct. 2001.

[5] 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 *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, 2003.

[6] J. Hill and D. Culler. Mica: A Wireless Platform for Deeply Embedded Networks. In *Micro, IEEE*, volume 22, pages 12–24, 2002.

[7] J. Hill, R. Szewczyk, et al. System Architecture Directions for Networked Sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2000.

[8] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MOBICOM '00)*, Aug. 2000.

[9] D. Johnson and D. Maltz. Dynamic Source Routing in Ad-Hoc Wireless Networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[10] P. Juang, H. Oki, Y. Wang, et al. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002.

[11] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002.

[12] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, June 2003.

[13] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

[14] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, Sept. 2002.

[15] Maxstream. 9XStream, wireless modem data sheet and OEM manual. `http://www.maxstream.net/`, June 2002.

[16] R. Min, M. Bhardwaj, S. Cho, N. Ickes, E. Shih, A. Sinha, A. Wang, and A. P. Chandrakasan. Energy-centric enabling technologies for wireless sensor networks. In *IEEE Wireless Communications*, volume 9, pages 28–39, Aug. 2002.

[17] Mpala Wildlife Foundation. Mpala research centre. `http://www.mpalafoundation.org/researchctr/`.

[18] Panasonic. CGR18650A, 2A-hour Lithium-Ion battery, cylindrical Model. `http://www.panasonic.com/`, Aug. 2003.

[19] C. E. Perkins and E. M. Royer. Ad hoc On-Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, Feb. 1999.

[20] Rockwell Science Center. Wireless integrated network sensors (WINS). `http://wins.rsc.rockwell.com/`.

[21] A. Savvides and M. Srivastava. A distributed computation platform for wireless embedded sensing. In *Proceedings of International Conference on Computer Design (ICCD)*, 2002.

[22] Sun Microsystems. Java 2 Platform, Micro Edition. `http://java.sun.com/j2me/`, Nov. 2002.

[23] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a Sensor Network Expedition. In *First European Workshop on Wireless Sensor Networks*, Jan. 2004.

[24] Texas Instruments. MSP430x1xx Family Ultra-Low-Power Micro-controller User's Guide. `http://www.ti.com/`, 2002.

[25] The Center for Conservation Biology. VAFALCONS. `http://fsweb.wm.edu/ccb/vafalcons/falconhome.cfm`, 2002.