Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems

Ting Liu
tliu@cs.princeton.edu
Department of Computer Science
Princeton University

Margaret Martonosi
mrm@ee.princeton.edu
Department of Electrical Engineering
Princeton University

ABSTRACT

Sensor networks are long-running computer systems with many sensing/compute nodes working to gather information about their environment, process and fuse that information, and in some cases, actuate control mechanisms in response. Like traditional parallel systems, communication between nodes is of fundamental importance, but is typically accomplished via wireless transceivers. One further key attribute of sensor networks is that they are almost always long-running systems, intended to operate in situ, with minimal direct human intervention, for months or years. This requirement for long-running autonomy mandates careful design of the runtime system that manages applications on each node, to ensure reliability and ease of upgrades over the life of the system.

This paper describes Impala, a middleware architecture that enables application modularity, adaptivity, and repairability in wireless sensor networks. Impala allows software updates to be received via the node's wireless transceiver and to be applied to the running system dynamically. In addition, Impala also provides an interface for on-the-fly application adaptation in order to improve the performance, energy-efficiency, and reliability of the software system. Impala has been designed to be a part of the ZebraNet mobile sensor network, but we are also prototyping it within HP/Compaq iPAQ Pocket PC handhelds. We present performance data for both real system measurements of the Pocket PC version as well as simulations of a full mobile sensor system deployment. Overall, Impala is a lightweight runtime system that can greatly improve system reliability, performance, and energy-efficiency. The ideas introduced here for sensor networks have applicability more broadly in other long-running autonomous parallel systems as well.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*Real-time and Embedded Systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and Interfaces*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific Architectures*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Systems; Real-time Systems and Embedded Systems*

General Terms

Design, Experimentation, Performance

Keywords

Sensor Networks, Middleware System, Software Adaptation, Software Update

PPoPP'03, June 11–13, 2003, San Diego, California, USA.

1. INTRODUCTION

In recent years, wireless sensor networks have emerged as a computing/communication domain of significant research interest. Comprised of many sensor nodes often distributed across the environment, wireless sensor networks represent a domain of parallel/distributed computing with heightened interest and importance in recent years. Sensors may capture data such as position or temperature, as well as images, auditory information and other inputs relevant to their specific intended use. Sensor nodes also have modest compute/communication resources to process data, for example to compress or encode it, to fuse the data with that collected by other local or remote sources, and to communicate it to other nodes in the network.

Because of the scale of sensor network systems, with typically tens, hundreds, or even thousands of nodes, coordinating the communication and computation across many nodes is complex. Depending on node topology, network connectivity, and node mobility, the communication protocol they use and even the computations they perform can vary widely. As such, it is nearly impossible for a single protocol to be appropriate all the time, even within a single sensor network application. Some amount of adaptivity is crucial for protocols to properly handle an interesting range of possible parameter values.

For these reasons, sensor networks need adaptive application software that can automatically discern needed parameter settings or software usages and adjust them automatically. Further motivation for adaptation comes from the fact that sensor nodes are typically operating with very tight computation and energy constraints; thus, highly-tuned operation may be the only way to solve a problem in the given energy budget. Finally, sensor networks are long-running systems in which device failures are overwhelmingly likely

to occur. Adaptation around failed devices becomes yet another pressing feature.

Adaptation of application software can be accomplished in several different ways. For example, one could build a monolithic piece of application software that is written to adapt to various scenarios envisioned at the time it was written. There are, however, several difficulties with monolithic adaptive software. First, these large adaptive systems can be hard to program properly, and so bugs are likely. Second, in addition to some pre-programmed adaptivity, it is also inevitable that software updates will be required during the months or years of designed-for lifetime of a sensor network deployment. Because sensors are typically deployed in large numbers in places that are inaccessible to system managers, an update and adaptation strategy needs to be based on broadcasting updates via the wireless transceiver. Impala's non-monolithic application structure aids with these wireless software updates, by allowing application updates to arrive in modular pieces, rather than requiring successful transmissions of very large shipments of new code.

This paper describes Impala, a middleware system and API for sensor application adaptivity and updates. Essentially, we propose a runtime system that acts as a lightweight event and device manager for each mobile wireless sensor node in the system. Impala has been built as part of the ZebraNet effort, in which sensing nodes are placed on free-ranging wildlife to perform long-term migration studies on a collection of animals in an ecosystem. The ZebraNet effort offers very clear motivation for remote software updates and adaptivity, since we clearly do not want to have to tranquilize and re-capture a collection of collared animals each time we have a software update to apply!

While on-the-fly software updates are used elsewhere, Impala is novel in examining the software architecture best suited for minimal performance and energy impact for code running on a sensor node that is both performance- and energy-constrained. It is also designed to handle the fact that in mobile networks with incomplete connectivity, a heterogeneous collection of protocol versions may be installed on various nodes at any point in time. The middleware encourages interoperability of these distinct protocols as long as it is safe. We feel that these middleware features are also valuable in a range of distributed and "grid" computing environments, so these results have fairly broad applicability.

As this paper shows, the Impala middleware layer introduces very little new overhead over monolithic approaches. We have built an Impala prototype on HP/Compaq iPAQ Pocket PC handhelds running Linux and report the event-handling overhead here. In addition, simulation-based studies on larger deployments show the efficiency of reprogramming the network with the application update mechanism and how the application adaptivity can help communication protocols discern higher-performing or lower-energy approaches for managing data in the ZebraNet system.

The remainder of the paper is structured as follows. Section 2 gives context for Impala by presenting a brief overview of the ZebraNet system. Section 3 then describes the Impala system overall. In Sections 4 and 5, we focus in on Impala's application adaptation and application update functionality. Section 6 offers an evaluation of the Impala system combining both real-system as well as simulation-based measurements. Finally, Section 7 sketches out related work and Section 8 offers our conclusions.

2. ZEBRANET PROJECT OVERVIEW

The Impala system is part of the ZebraNet project [10], a mobile sensor network system aimed at improving tracking technology via energy-efficient tracking nodes and peer-to-peer communication techniques. 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.

Sensor networks like ZebraNet are a form of parallel system, since they involve the coordination of tens or hundreds of compute devices. While autonomic and adaptive computing is a common research theme in many systems today, its motivation in Impala comes quite directly from the anticipated usage of ZebraNet. We aim to deploy 30 or more ZebraNet nodes as tracking collars in long-term operation. Over the course of the study, software upgrades and adaptation are needed. Manual upgrades are difficult, however, since the compute nodes are on wild animals who would need to be tranquilized to retrieve the nodes and change software manually.

A ZebraNet hardware node includes global positioning system (GPS), a simple microcontroller CPU, a wireless transceiver, and 1-8MB of non-volatile storage to hold logged data until it can be transmitted elsewhere. ZebraNet does not count on constant communication access to a base station, but instead uses periodic node discovery and peer-to-peer communication to communicate data towards the base station by using other peer nodes for store-and-forward routing. Rather than a connection-oriented scheme in which a node identifies a full route to the base, the data is instead transmitted to base hop by hop via other nodes, with heuristics that guide the data to where the base was last seen.

In our first version of ZebraNet, the main "application" software running on each node is the communication protocol software trying to get the data back to the base station. But in future sensor systems, more complex application software would also be filtering and fusing sensor data, in addition to communicating it. Impala is designed to support a range of applications.

3. IMPALA: A MIDDLEWARE ARCHITECTURE FOR ZEBRANET

In energy-constrained systems like a sensor node, it is all too tempting to build monolithic but lean software, hard-coded to handle exactly the envisioned compute needs of the node, and no more. These monolithic approaches are appealing at first glance because they can make the best use of the meager compute resources available, and they do so with the smallest possible energy consumption.

Typically however, when making a case for modular code, the long-term view is important. In particular, many sensor networks will be deployed in harsh surroundings, where they are intended to run without user intervention for months or years at a time. Some sensor networks are designed to have hundreds or thousands of nodes. Some sensor networks are distributed over huge geographical areas. For reasons like this, designers must consider the long-term management of the sensor application software as an integral part of the design process. By adopting a middleware layer that can update and adapt applications dynamically, new protocols can be plugged in at any time, and switches between pro-

tocols can be performed at will. The Impala approach we outline here uses a middleware layer that is intended to act as an operating system, resource manager, and event filter on top of which specific applications can be installed and run.

The major contributions of our work are:

- We explore the implementation of a non-VM-based middleware layer intended for infrequent code updates.
- We link the functionality for application adaptivity and application updates, offering a framework that works well for both.
- We evaluate a full system prototype of our approach, in addition to simulating more large-scale sensor deployments of Impala.

3.1 Design Rationale

While combining multiple application protocols into a large, adaptive and self-updating protocol is possible, our layered approach has several advantages over the monolithic approach.

- **Modularity:** With a middleware layer taking care of the switching decisions, applications can be independent and do not need to coordinate with each other. With the middleware layer also handling the update issues, applications can focus on their objectives and disregard the need for contemporaneous software execution and update.
- **Correctness:** Impala makes application correctness easier to achieve because programming individual applications is simpler than programming a super-application with many interacting and updating components.
- **Ease of Updates:** Software changes such as adding, deleting and modifying an application can be simpler because they can involve only local code changes within a module. By contrast, in a monolithic approach, even small changes may have global repercussions elsewhere in the code.
- **Energy Efficiency:** Rather than transmitting an entire monolithic program, software updates can be transmitted at the granularity of smaller program module. Since the network transmitter is the most power-hungry component in an energy-constrained sensor network, this offers significant energy savings.

3.2 System Architecture

Figure 1 shows the system architecture of Impala. The upper layer contains all the application protocols and programs for ZebraNet. These applications use various strategies to achieve a common task of gathering the environment information and routing it to a centralized base station via peer-to-peer transmission. Only one application is running at a time.

The lower layer contains three middleware agents: the Application Adapter, the Application Updater, and the Event Filter. The Application Adapter adapts the application protocols to different runtime conditions to improve performance, energy-efficiency and robustness. The Application Updater receives and propagates software updates through

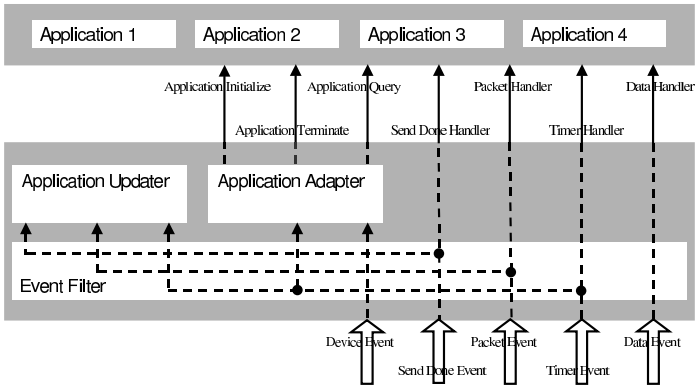

Figure 1: Layered system architecture.

the wireless transceiver and installs them on the node. The adapter and updater are described in detail in Section 4 and 5.

The Event Filter captures and dispatches events to the above system units and initiates chains of processing. Impala has five types of events.

Timer Event signals that a timer has gone off. Impala has three timers owned, respectively, by the current active application, the Application Adapter, and the Application Updater. The owner of the timer handles these events.

Packet Event signals that a network packet has arrived. Impala has two types of packets, application-to-application packets and updater-to-updater packets. The intended receiver of the packet handles these events.

Send Done Event signals that a network packet has been sent or has failed to send. It allows asynchronous network transmission. The original sender of the packet handles these events.

Data Event signals that a data sample from the sensing device is ready to read. The current active application handles these events.

Device Event signals that a device failure is detected. The Application Adapter handles these events.

When multiple events arrive at the same time, they are processed sequentially. This eliminates the programming complexity of synchronizing between different event handlers. To prevent the significant delay in processing subsequent events, however, all event handlers are required to complete within a limited amount of time. Therefore, any blocking operation such as network transmission must be handed over to other system components and performed asynchronously.

3.3 Application Programming Model

Figure 1 illustrates the event-based programming model of Impala. The applications, the Application Adapter, and the Application Updater are all programmed into a set of event handlers which are invoked by the Event Filter when the associated events are received.

In particular, applications must implement four event handlers: `timer handler`, `packet handler`, `send done handler`,

and `data handler`. In addition, to help the adapter query the applications and switch from one to another, the applications are also required to implement three other routines: `application query`, `application terminate`, and `application initialize`.

The Impala user library contains a number of general programming utilities. Networking utilities allow applications to send asynchronous messages in encapsulating packets and, upon completion, generate a `send done` event to the event filter. Timer utilities allow applications to set up a timer for various purposes, for example, to send messages at a certain time or at regular intervals. A timer can be set, reset and canceled over and over again. Device utilities allow applications to have a certain degree of control on hardware devices, for example, to turn on and off the network transceiver. These programming utilities are also available to the adapter and updater.

The global data structures include a uniform storage image of the sensed data and an execution frame to save the application execution states. First, the storage of the sensed data is a stateful resource across applications, as changes made to the storage by one application will be passed to another. Therefore, all applications must agree upon the basic storage organization. Nevertheless, the specifics of the storage utilization are up to individual applications. The uniform storage image defined in Impala is a list of data generated from local sensing device and a list of data received from other nodes and a log of data that has been successfully routed to the base station. Second, every application needs an execution frame shared among its event handlers to save execution states for network communication, memory management, etc. Impala defines a single execution frame for all applications. Its format, however, is application-specific.

To sum up, Figure 2 shows a timeline example of the event-based application programming model. The timelines show a series of events and event handler actions between two communicating sensor nodes in a two-hour data communication cycle in ZebraNet. The arrows indicate when things happen and the text planes show what happens in each layer. Application event handlers respond to application events to perform routing operations such as peer discovery and data transfer, and application routines are called by Impala event handler to help query and switch applications.

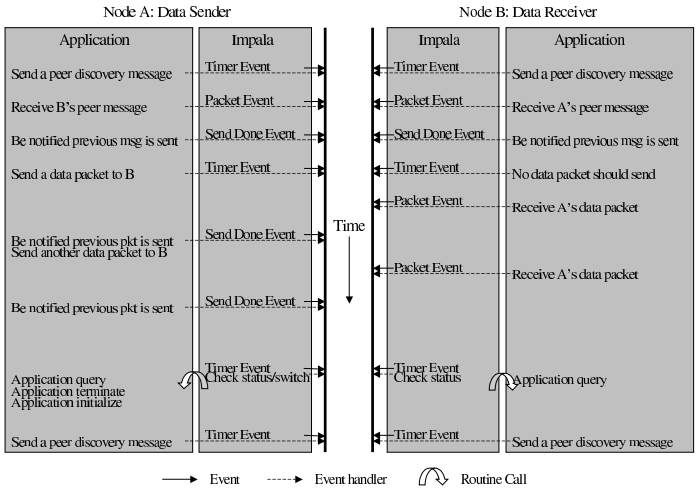

Figure 2: Timeline example of event-based application programming model.

4. APPLICATION ADAPTER

In ZebraNet, sensor nodes are loaded with multiple applications for routing data back to the base station, with different of these applications applicable under different conditions. Adaptation is desirable in two scenarios. First, as performance, energy-efficiency, and other attributes of a protocol depend heavily on a number of factors, the adapter handles an interesting range of parameters and adapts to sensitive changes in their values. Second, often, a device that is critical to some protocols is not critical to others, therefore the adapter adapts protocol choice to hardware failures to improve robustness.

Adaptation is implemented through Impala's event-based programming model, and occurs in response to a range of events. Some events, like timer events, signal that time has passed since the last status check; the adapter may then choose to query application or system states in order to determine if any adaptation should be performed. Other events, like device events, have sources external to Impala and signal an external event for Impala to respond to, such as the failure of a particular radio transceiver; the adapter should then examine the impact of the failure and determine whether to dispatch another application to work around it.

Because the adapter has the best overview of the runtime states, it is our design choice to let the adapter rather than applications make switching decisions, although the alternative could yield less restrictive programming model or more flexibility in application design. Currently our adaptation is mainly based on local states of individual sensor nodes, although we eventually also plan to adapt based on global coordination of what is best for the overall sensor network.

4.1 Adapter Functionality

A set of application parameters and system parameters is defined to represent the runtime states. Application parameters comprise information known only by a particular running application, while system parameters represent information known by Impala.

Application parameters might include recent histories, averages, or totals of: (i) the number of direct network neighbors encountered, (ii) the amount of sensor data successfully transferred to peer sensor nodes, (iii) the amount of free storage for application data, and so on. System parameters include: (i) battery level, (ii) transmitter range/power/datarate, and (iii) the geographic position of the node.

Each application is specialized in tracking a subset of the application parameters and is responsible for reporting their values. The adapter has an Application Parameter Table that records which application tracks which parameters. Our current implementation allows 64 application parameters, so the table size is quite reasonable. This table is used in application query and switch.

In order to catch any sensitive parameter changes, periodically, the adapter queries the current active application for the parameter values it claims to track, fetches the system parameter values, and examines the switching rules. If a switching rule is satisfied, an application switch will be performed. As some application parameters are histories of attributes, such as the average number of direct network neighbors over the last k cycles, the adapter also passes these

```
1. for each rule associated with the current active application
2.     for each parameter in the rule with unknown value
3.         if the parameter is an application parameter
4.             query the application for the parameter value;
5.         else
6.             get the parameter value by a system call;
7.     if the rule is satisfied by the parameter values
8.         query the application for unknown parameter values;
9.         terminate the current active application;
10.        pass the parameter values to the next application;
11.        initialize the next application;
12.        return;
```

Figure 3: Pseudo-code for application query and switch.

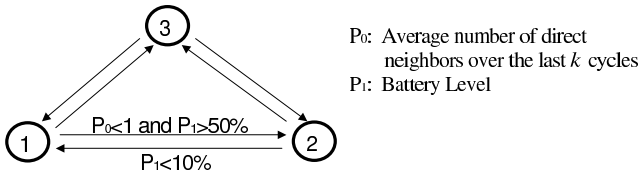

Figure 4: Adaptation Finite State Machine.

parameters to the next application after the switch.

Figure 3 shows the pseudo-code of application query and switch. As every sensor node in ZebraNet is active for network communication for half an hour in every two hours, we choose to conduct the application query at the end of the network activity period as is shown in Figure 2. Thus, the adapter will not interfere with application communications, or reduce network bandwidth which is precious in our sensor network.

Having queried parameter values, the adapter makes adaptation decisions by examining the Adaptation Finite State Machine (AFSM) which is shown in Figure 4. States in the AFSM correspond to different applications. Arrows represent adaptive transitions from one application to another, and the parameter expression above each arrow is the condition under which the switch happens.

For example, assume Protocol 1 is a very selective protocol that uses the short-range radio to transmit to a single neighbor, while Protocol 2 is a more indiscriminate flooding protocol that uses the long-range radio to transmit. Protocol 1 consumes less energy and generates less network traffic, but it only performs well when nodes encounter each other for transfers fairly frequently. In contrast, when a sensor node is isolated at a remote location but has enough energy to spend, Protocol 2 may allow it to connect with others more effectively. Therefore, we adopt an adaptive switching rule that says "*Switch from protocol 1 to protocol 2 if the average number of direct neighbors has been less than 1 for the last k cycles and the battery level is above 50% of the full charge*". Likewise, there should be another switching rule that says "*Switch from protocol 2 to protocol 1 if the battery level is below 10% of the full charge*".

4.2 Device-based Adaptation

Adaptation to device failures is accomplished by a method very similar to the parameter-based adaptation already described. Some device failures will cause device events which can be responded to. Other device failures will be discovered through periodic queries that discover a non-responding piece of hardware. In order to respond effectively to a device failure, Impala has an Application Device Table that records which application relies on which hardware devices. In our prototype, we size the table to allow eight devices to be tracked, although we currently only have three instantiated in the results we present here: a short-range radio, a long-range radio, and the node's GPS transceiver. Upon a device failure, the adapter will respond by disabling the protocols that requires the failed device. If the current active application is disabled, it will switch to an application that does not require the failed device.

5. APPLICATION UPDATER

Although a similar software update problem was previously studied in Maté [12], ZebraNet and many other sensor networks have distinctive characteristics that change the design tradeoffs:

- **High Node Mobility:** The sensor networks targeted by Maté are comprised of static nodes. In ZebraNet, however, nodes have fairly high mobility and move in clustered patterns.

- **Constrained Network Bandwidth:** In ZebraNet, data is frequently collected by the sensing devices and must all eventually be transmitted to the base station. This generates significant network traffic and makes the network bandwidth available for software transmission even more constrained.

- **Wide Range of Updates:** ZebraNet software will be updated in a wide range, from small bug fixes to major application enhancements, and even to adding and deleting complete applications.

These characteristics have implications on several issues the Application Updater must handle.

- **Incomplete Updates:** Because network bandwidth is low and network connectivity can be broken when nodes are moving around, it can take several attempts for a node to gather the elements of a complete update from the network. Therefore, incomplete updates will be common.

- **Update vs. Execution:** It is impractical to halt execution and wait for an update to complete as in Maté as it may take long. Therefore, in-progress software updates must be processed in parallel with software execution.

- **Contemporaneous Updates:** The network infection time of an update is potentially long. Therefore, when updates are issued close to each other, nodes may receive multiple incomplete updates, and some of them may be out of order.

- **Inconsistent Updates:** In Maté, each application module is updated independently assuming the update is compatible with existing modules. In ZebraNet, however, the potentially dramatic software changes mean that modules must be associated with the correct version of software.

- **Propagation Protocol:** The updater must find an effective communication protocol that not only propagates the software rapidly but also consumes the minimal network bandwidth. The persistent broadcasting method used in Maté is not affordable in ZebraNet.

- **Code Memory Management:** In ZebraNet, sensor nodes will be loaded with four applications each of which contains seven program modules. With this large amount of software existing in a sensor system, code memory management will be challenging.

Overall, the goal of the Application Updater is to achieve an effective software update mechanism for mobile wireless sensor networks with resource constraints.

We designed a software management and transmission mechanism for dynamic software updates as summarized below. For software management, we store both complete and incomplete update versions in the code memory. Complete updates are logged for execution. Incomplete updates are logged so that they will be able to resume from last time should the update be available again. For software transmission, we adopt an on-demand transmission strategy. Sensor nodes periodically exchange software version information before exchanging the actual code and will do so only if requested. The frequency of the version information exchange is automatically adjusted based on an estimate of whether all sensor nodes have the most recent updates. The following sections describe our design in detail.

5.1 Software Compilation, Linking, and Memory Layout

Before injected into the network, a program module is compiled into binary instructions. Linking is performed by the updater on every sensor node. A module will not be linked to the main program until all the modules in the same update have been received. We do not allow cross references between modules. This means one module's linking is independent of other modules. Therefore, modules that are already linked and will be re-used in later versions do not have to be re-linked. After linking, the new application is considered "installed" on the node.

The code memory stores both complete and incomplete applications. An application is complete if all its modules have been received. Over time, we keep the highest complete version and several higher-but-incomplete versions for each application. Memory space is dynamically allocated for an incoming module and deallocated for a relinquished one. The target size of a module is 4KB in average and 8KB at the maximum. Each module occupies a consecutive memory space aligned at 2KB. This means the smallest allocatable memory block is 2KB and the biggest allocation request is four such blocks. Figure 5 shows the code memory layout.

5.2 Software Version Numbers

Impala adopts a module-based version system to facilitate long-term software development and update. Each module has a version number, and each application as a whole has a version number. Version numbers increase monotonically.

The module-based version system allows selective software transmission. Before exchanging software updates, nodes first exchange an index of application modules and then only request the changed modules for transmission. This prevents the transmission of unchanged modules and therefore saves network bandwidth.

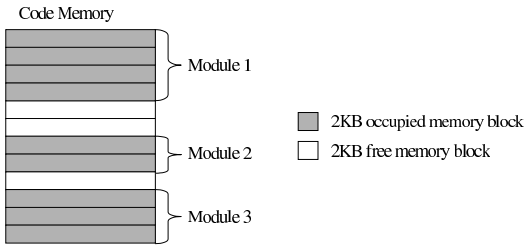

Figure 5: Code memory layout.

5.3 Software Transmission

We adopt an on-demand software transmission strategy with three stages. Figure 6 shows its pseudo-code. At the beginning, every node advertises its software version information to others so that everyone knows who has which versions. The software version information includes the module-based version numbers of the complete applications and the highest application version numbers known so far. For nodes with newer applications, this information is an implicit offer of software updates. For nodes with older applications, this information is an implicit request for software updates. Since we want to propagate software updates as quickly as possible and still preserve network bandwidth, the software version information should be advertised if and only if all nodes do not have the complete most recent updates.

After receiving the software version information from network neighbors, a node will determine what is the highest complete version available for each application and who has it; it will send out requests for modules it does not have. A node may already have some modules of the newer version, therefore making explicit requests saves unnecessary software transmission.

In the final stage, the node with the highest complete version will transmit the actual code requested by its neighbors. Ties are broken by a node's ID which is unique throughout the network.

The three-stage procedure will repeat on each node if it finds out that, before the last stage, not everyone in its network neighborhood has the complete applications that are the most recent known so far, because this is a good indication that the population of the new software has not converged. Otherwise, it will exponentially back off the repeat timer. This timing control will automatically maximize the software propagation speed at the initial propagation phase and automatically slow it down towards the end. It also saves unnecessary broadcasts of software version information if the population of the new software has converged or if it has not been propagated to the local area network.

5.4 Software Reception and Installation

When an application update is received from the network, it is logged in the Application Update Lists (AUL) as is shown in Figure 7. Each node in the lists is an Application Update Record (AUR) whose format is shown in Figure 8. Entry pointers in the AUR locate the program modules in the code memory. They can be `null` if the corresponding modules have not been received. AURs in the same list are sorted by their application version numbers.

```
1. if in stage 1
2.     send my software version information;
3. else if in stage 2
4.     for each application
5.         check the version information collected in stage 1
6.         if a higher version is found
7.             check which modules I need;
8.     send a request for all the modules needed
9. else if in stage 3
10.    for each application
11.    check the requests collected in stage 2;
12.        if this application is requested from me
13.            send the first packet of the first requested module;
```

Figure 6: Pseudo-code for on-demand software transmission with three stages.

As it may take a long time to receive an update completely, the updater tries to keep records on every incomplete update so that software improvement, even if intermediate, can be applied whenever possible. This will also save network bandwidth should the update show up later. However, keeping every incomplete update is expensive in terms of memory and energy consumption. If there is insufficient memory space due to too many incomplete updates, some can be released to make space. The release goes from the lower versions to the higher versions, and from the same application to other applications.

When the system receives an update, an AUR is added to the AUL. If the software reception is interrupted, the AUR will record which modules are still missing and need to be re-transmitted next time they are available. Modules are considered missing if not received in their entirety.

When software reception is complete, the updater can install the new application on the node. First, to guard against unexpected programming errors, the updater performs simple security checks. Such errors include invalid memory accesses and the absence of a return statement within a module that causes the execution to step beyond the boundary. We do not yet, however, attempt to protect against malicious programs. If this eventually proves to be desirable, we will consider using methods such as the proof carrying code.

Second, linking is performed on the unlinked modules. Finally, the updater copies the AUR to the Application Activation Table (AAT). The format of an entry in the AAT is shown in Figure 9. It is similar to an AUR except that the entry pointers are all defined. The AAT serves as an application link table that links the applications to launch to the main program. The event filter will use it to invoke the application event handlers and the adapter will use it to call the application routines. The AAT also serves as a software version information table that will be advertised in the first software transmission stage. After being copied to the AAT, the AUR as well as the AURs with lower application version numbers will be removed from the AUL. No memory copy for software is needed because both the AUR and the AAT use pointers to the memory location of the actual code. If the current active application is being updated, the updater will terminate the old version, overwrite the AAT entry, and initialize the new version.

The Application Updater is also implemented through Impala's event-based programming model as is illustrated in Figure 10. The example shows two updaters performing software transmission, reception, and installation in a two-hour software communication cycle in ZebraNet.

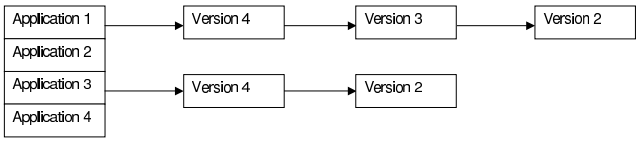

Figure 7: Application Update Lists.

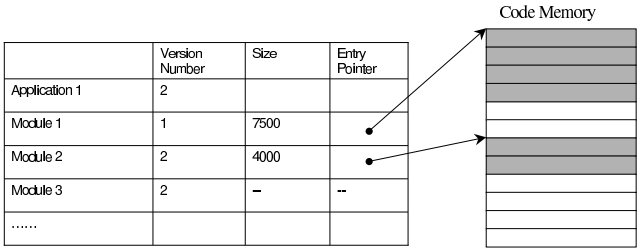

Figure 8: Application Update Record.

6. SYSTEM EVALUATION

6.1 System Implementation and Overhead

In addition to achieving the middleware functionalities, Impala is also aimed at a lightweight system layer. To help evaluate its overhead in real sensor networks, we prototyped Impala on modern mobile computing devices, the HP/Compaq iPAQ Pocket PC handhelds, and measured its overhead in event delivery and processing.

6.1.1 Implementation Details

The mobile devices we used for Impala prototypes had the following attributes. We used a pair of iPAQ handhelds with 206MHz CPUs running Linux Familiar 5.3 and the Xipaq GUI [5]. Each handheld included 32MB Flash RAM as the main memory and 16MB flash ROM as the file system.

For the Impala middleware layer, we implemented the Application Adapter, the Application Updater, the Event Filter, and the networking and timer utilities. For the application layer, we implemented two application protocols, a very selective history-based protocol and a more indiscriminate flooding protocol. Table 1 shows the memory footprint of the application programs.

The experiment was performed on two iPAQ nodes both of which ran Impala for three one-minute cycles. Each experiment cycle is equivalent to a two-hour data/software communication cycle in ZebraNet. Both nodes are loaded with both protocols. The flooding protocols are of the same version, but the history-based protocol on node 0 has a newer version of `packet handler` than that on node 1. Both nodes start with the history-based protocol. In the first cycle, the newer packet handler will be transmitted from node 0 and installed on node 1. But no sensor data will be transmitted as neither node considers the other as an appropriate transmission target based on the history-based protocol. In the second cycle, both nodes will switch to the flooding protocol because in the previous cycle they found too few direct network neighbors for which the history-based protocol does not perform well. Therefore data exchange will happen. The updaters will see consistent application versions this time, therefore software transmission will not happen and the updaters' timer will back off by 1 cycle. In the last cycle,

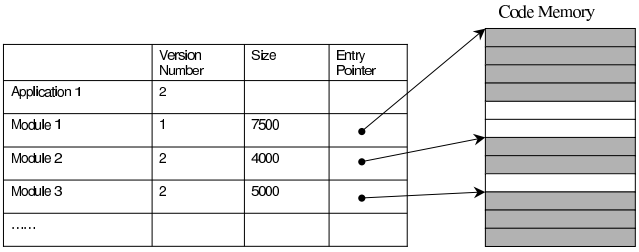

Figure 9: Application Activation Table.

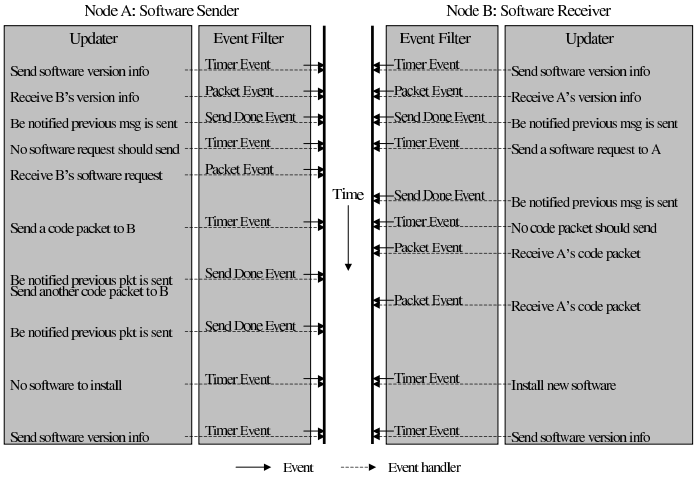

Figure 10: Event-based software transmission, reception, and installation.

both nodes switch back to the history-based protocol because flooding has achieved enough data distribution. This time, node 1 will run the newer version of the history-based protocol. The updaters will stay idle.

6.1.2 Overhead Measurement

Our measurement is focused on Impala's overhead in event delivery and processing. As opposed to a monolithic approach that hands over application events directly to the application layer, Impala delivers events after checking their destinations, which delays the processing. It turns out, however, the event delivery latency for `packet events`, `send done events`, and `timer events` only involves one branch compare instruction. And since `data events` are handled only by the current active application, no destination check, and therefore, no latency is involved.

Impala also processes events for itself. Some event processing time is inherent to Impala and not applicable to the monolithic approach, for example, the time for application query and switch. Other event processing time is commonly required, for example, the time for software transmission, reception and installation. Table 2 lists the processing time for both Impala events and application events to show the relative values. In general, however, Impala events occur less frequently than application events. In Table 2, the time to receive and process a code packet by the updater is much longer than others. This is because, in our prototype implementation, the received code packets are written to files in the flash ROM of the iPAQ which is extraordinarily slow. In fact, the write operation takes 95% of the total time.

Program Modules in History-based Protocol	Instruction Size (Bytes)	Program Modules in Flooding Protocol	Instruction Size (Bytes)
Data Handler	1104	Data Handler	1044
Timer Handler	2228	Timer Handler	1752
Packet Handler	808	Packet Handler	560
Send Done Handler	1360	Send Done Handler	1360
Application Initialize	1140	Application Initialize	568
Application Terminate	28	Application Terminate	28
Application Query	360	Application Query	108

Table 1: Memory footprint of application programs.

6.2 Efficient Network Reprogramming

An important design goal for Impala is to achieve the optimal software propagation speed while consuming the minimal network bandwidth. To demonstrate the efficiency of Impala's on-demand software transmission strategy, we conducted simulations to reprogram the entire sensor network. In the experiments, 50 sensor nodes move around in a 40km by 40km square map, and software updates are broadcasted from the base station to every node in range and further propagated via peer-to-peer transmission. The radio range of the nodes is 4000 meters. The updaters on the nodes wake up once every two hours to exchange software updates. Two update strategies are evaluated. The first one is to broadcast the updates to other nodes with a probability every time the updater wakes up. Updates will be discarded if they are received by nodes that have more recent updates. The second one is our on-demand software transmission strategy.

Figure 11 shows the time for an update to infect the network. Time 0 represents the point when the base station first shows up in the network with the update. When the probability equals 1, the probabilistic broadcasting strategy achieves the maximal propagation speed, which is presented as the ideal case. Figure 11 shows that, with our on-demand software transmission strategy, the network infection rate increases rapidly at the beginning and slows down towards the end. It closely follows the ideal curve and outperforms the probabilistic broadcasting with lower probability. There are cases, however, where our strategy misses immediate update opportunities. For example, when a node from a cluster of updated nodes encounters a node from a cluster of unupdated nodes, both nodes think the updating procedure has converged and back off their timers of software version information advertising since they both have seen homogeneous population of updates. In that case, the two nodes can not discover each other until the end of the back off interval, which delays the update.

Figure 12 shows the number of software transmissions performed to infect the network. For our on-demand transmission, it increases approximately with the number of updated nodes. For the probabilistic broadcasting, it is much higher and increases linearly over time. The difference in the actual network bandwidth consumed by the two strategies could be even larger. This is because our strategy only selects the updated modules to transmit, while the other strategy does not have any discrimination. Our strategy, however, has a communication overhead which is the software version information and software request messages. When the net-

Processing Categorization	Processing Details	Processing Time (us)
Application Execution	Peer discovery message construction and transmission	3211
	Peer discovery message reception and processing	14
	180-byte data packet construction and transmission	3563
	180-byte data packet reception and processing	38
Application Adaptation	Application query and switch	1301
Application Update	Software version information construction and transmission	478
	Software version information reception and processing	102
	Software request construction and transmission	1230
	Software request reception and processing	58
	256-byte code packet construction and transmission	2024
	256-byte code packet reception and processing	12617
	Software installation	2329

Table 2: Impala event processing time.

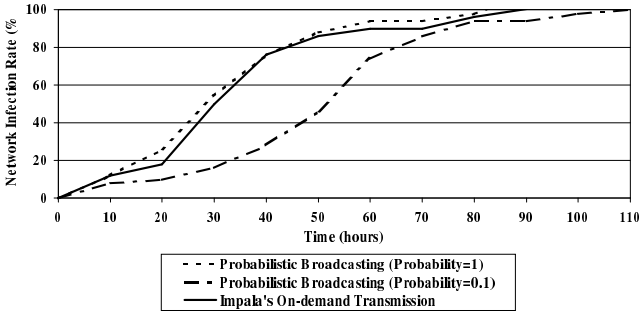

Figure 11: Network infection rate.

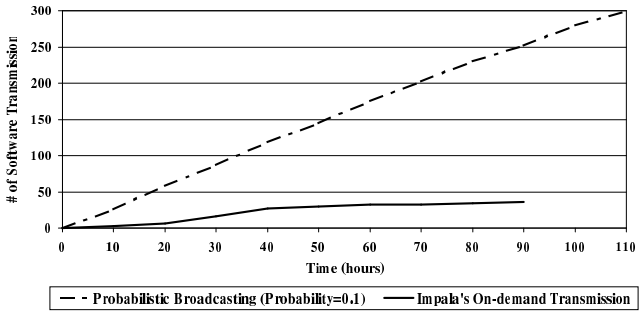

Figure 12: Software transmission volume.

work reprogramming is complete, 247 version messages and 47 request messages are transmitted in addition to 36 actual software transmissions.

6.3 Potential Benefits of Adaptation

Another important design goal for Impala is to achieve "intelligent" scheduling among various protocols to adapt to different conditions and ultimately improve the overall system performance. To demonstrate the effectiveness of protocol adaptation, we conducted several experiments focusing on improvements on three system aspects, routing performance, energy efficiency, and robustness.

6.3.1 Simulation Methodology

The experiments were conducted on an upgraded version of our sensor network simulator ZnetSim. Details about the original ZnetSim simulator can be found in [10]. Briefly, 50 sensor nodes move around in a 40km by 40km square map. Each sensor node has two radios, a short range radio and a long range radio. Several routing protocols are programmed in the network. The direct transmission protocol is a baseline protocol and does not involve any peer-to-peer transfer. Each node only transfers data to the base station if it is in range. The flooding protocol sends data to every network neighbor. The history-based protocol encodes the likelihood of a node being in range with the base station by assigning each node a hierarchy level based on its past success at transferring data to the base station, and selects a neighbor with the highest hierarchy level to send data to. Each simulation simulates the sensors' activities in 30 days, and computes the routing success rate which is the percentage of data ultimately received by the base station.

6.3.2 Example 1: Improving Routing Performance

Our first experiment demonstrates how protocol scheduling can tune the sensor's communication pattern for different network connectivities and achieve the optimal routing success rate. In this experiment, each sensor node can schedule two protocols, the flooding protocol and the history-based protocol, using the same radio device whose radio range is varied from 100 meters to 9100 meters. The two dotted curves in Figure 13 show the base scenarios where all sensor nodes always run the same protocol. When the radio range is low, the network connectivity is poor and sensors can hardly find a neighbor. In this case, the flooding protocol is better than the history-based protocol because by sending data to every neighbor it maximizes the chance of finding a data path towards the base station. However, when the radio range is high, the network connectivity is good and sensors can transmit only when other sensors in the vicinity are not. Therefore, in this case flooding is outperformed by history-based because by injecting too much data into the network it becomes constrained by the limited network bandwidth. Based on this fact, we apply the following switching strategy to each sensor to adapt to the

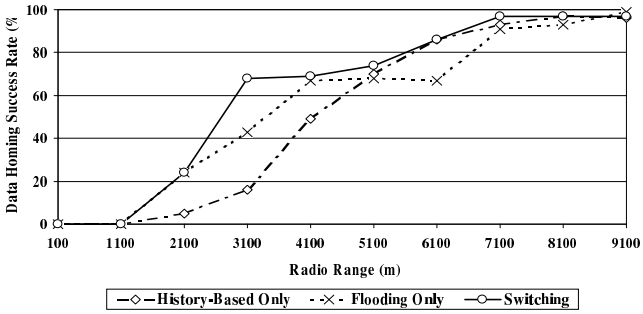

Figure 13: Routing performance improvement by protocol switching.

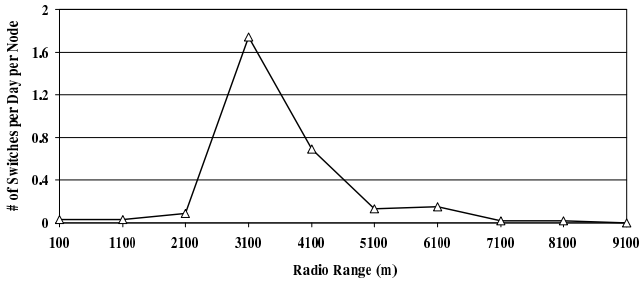

Figure 14: Protocol switching frequency.

network connectivity and hence bandwidth availability:

- At the beginning, run the history-based protocol
- If in the history-based protocol, and the average number of direct network neighbors encountered in the last half-hour protocol communication window is less than 16 and the average number of indirect network neighbors is less than 42, switch to the flooding protocol
- If in the flooding protocol, and the average number of direct network neighbors is more than 20 or the average number of indirect network neighbors is more than 42, switch to the history-based protocol.
- Otherwise, stay in the same protocol

The solid curve in Figure 13 shows the outcome of our switching strategy. For radio ranges that are very low or high, it always follows the better of the two non-switching curves. For some medium radio ranges, the switching curve is even better than both non-switching curves. This further demonstrates that when the network connectivity is hybrid with some nodes clustered together and others at remote distances, our protocol scheduling is able to pick up the appropriate routing protocol for a specific local network condition.

Figure 14 shows the frequency of protocol switching in this experiment. At 3100 meters, where protocol switching is the most frequent and the most performance improvement is achieved, the switching cost is, in average, less than twice per day for a sensor node.

6.3.3 Example 2: Improving Energy-Efficiency

In this experiment, each sensor node schedules between two protocols to improve energy-efficiency. The two protocols considered are the history-based protocol using a 1000-meter short range radio to transmit and the flooding protocol using a 5000-meter long range radio to transmit.

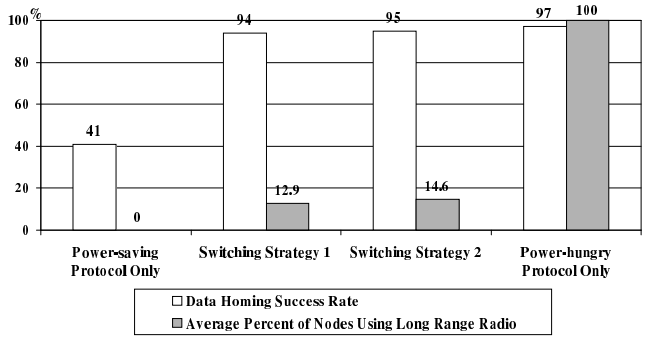

Figure 15: Energy-efficiency improvement by protocol switching.

The first protocol not only has a low-power radio, but also is conservative in data transmission, since sensors only send their data to nodes with better track records than themselves at delivering data to the base. In contrast, the second protocol employs a powerful radio and aggressive data transmission; it has robust routing performance but poor energy-efficiency. Clearly the protocols have complementary attributes: it is desirable to use the first protocol for energy reasons, but to adapt to the second protocol as a supplement when data delivery success rates have dropped too low.

For each sensor node, we adopt the following switching strategy:

- At the beginning, run the history-based protocol
- If in the history-based protocol, and the average number of direct network neighbors encountered in the last 8 hours is less than *minpeers* where *minpeers* is an adjustable parameter or the sensor node has not transmitted to either a peer sensor node or the base station for more than 12 hours, switch to the flooding protocol
- If in the flooding protocol, and it has transmitted to at least 1 peer sensor node or to the base station, switch to the history-based protocol.
- Otherwise, stay in the same protocol

Figure 15 shows the result of our switching strategy. We set *minpeers* to 1 in strategy 1 and 4 in strategy 2. It is clear that with protocol scheduling, we can improve the routing performance of the power-saving protocol to almost as good as that of the power-hungry protocol while using it only less than 15% of the time. Note that unlike the previous experiment, we assume that the network bandwidth is unlimited and will not be a constraint for any protocol.

6.3.4 Example 3: Improving Robustness

This experiment tries to demonstrate that protocol scheduling can circumvent device failures and improve system robustness. In this experiment, each sensor node can schedule four protocols, the history-based protocol using a 1000-meter short range radio to transmit, the flooding protocol using a 5000-meter long range radio to transmit, and the direct transmission protocol using either the short or the long

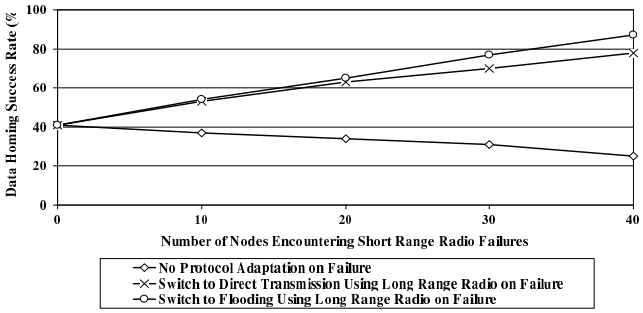

Figure 16: Switching on short radio failures (infinite battery charge).

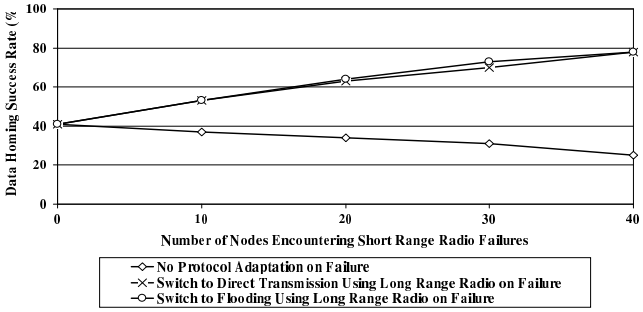

Figure 17: Switching on short radio failures (limited battery charge).

range radio to transmit. We randomly select a number of sensor nodes to encounter a radio device failure and let the failures occur at random time. When a radio fails, a sensor node will switch to a protocol using the other radio. The failed radio will not be able to receive or transmit data afterwards. This implies no data exchange between a pair of sensor nodes if the sender is transmitting over a radio that fails on the receiver side although the receiver has the other radio listening.

Since we tend to use a power-saving protocol as the major protocol and a power-hungry protocol as a supplement, we investigate a typical scenario where all sensor nodes run the history-based protocol at the beginning and switch to a protocol using the long range radio upon a short radio failure. As is shown in Figure 16, the routing success rate increases with the number of failures. This is because the protocol switching increases the network connectivity and the communication frequency at the expense of much more energy cost. Therefore, we limit the battery charge rate so that sensors will run out of battery when more and more sensors switch to the flooding protocol. The result is shown in Figure 17.

7. RELATED WORK

In sensor network studies, the most salient related work for Impala is Maté [12], a tiny virtual machine on sensor nodes. Maté is a bytecode interpreter that runs on TinyOS [7], a specialized operating system designed for motes [11]. Maté helps sensor network programmers to build expressive and concise applications, and it protects the system from being crashed by applications. It performs infection of small program capsules throughout the network by broadcasting. Although Impala has some resemblance to Maté in software update, there are some key differences between them. First, the sensor system model and therefore the software update problem that Impala has to deal with is quite different from Maté. Second, Maté integrates the propagation and installation of software updates in the programming of applications. By using a separate system layer performing these tasks, Impala lowers the complexity and enhances the modularity of application programming. Second, Maté will have energy savings if software update is fairly frequent (once every six days). However, in ZebraNet, we envision the software update to happen only at the frequency of once every couple months. Third, Maté can prevent applications from crashing the system, while Impala's security check is more oriented to unfortunate programming errors than malicious attacks.

Impala manages system resources and events and acts similarly as an operating system on sensor nodes. Therefore, studies on sensor network operating systems such as TinyOS [7], Micro Amps [19], and Magnet OS [1], and studies on sensor network architectures such as the Smart Dust [11], Pico Radio [13], WINS [17] and SCAADS [6] projects provide us with design paradigms of system architecture and programming model. In particular, our event-based programming model is motivated by TinyOS. Nevertheless, the key difference between Impala and TinyOS is that the event-based model of Impala is specially tailored for the development, management, and update of applications while the event-based model of TinyOS is more generalized for the interactions of system components across different system levels.

Finally, Impala is envisioned to support a broad range of applications for ZebraNet. This requires the understanding of the programming methodologies and needs of real sensor network applications. Therefore, our design were also guided by studies on sensor network applications and routing protocols [3], [8], [9], [16].

8. CONCLUSION

This paper presents Impala, a middleware architecture that enables application modularity, adaptivity, and repairability in wireless sensor networks. Impala supports multiple applications by adopting an event-based modular programming model and providing a friendly programming interface. Impala uses a lightweight system layer to perform dynamic application adaptation based on parameters and device failures and automatic application updates based on a specialized software management and transmission mechanism. Through both a prototype implementation on HP/Compaq iPAQ Pocket PC handhelds as well as simulations of a full deployed mobile sensor network, we show that the Impala middleware architecture has low overhead, achieves efficient network reprogramming, and can offer effective improvements on the performance, energy-efficiency and robustness of many long-running sensor systems. More broadly, we feel that Impala offers insights that can improve the behavior of other long-running "autonomic" parallel systems as well.

9. REFERENCES

[1] R. Barr, J. C. Bicket, D. S. Dantas, et al. On the Need for System-level Support for Ad-Hoc and Sensor Networks. *Operating Systems Review*, Apr. 2002.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In

Proceedings of the 13th ACM Symposium on Operating Systems Principles, Oct. 1991.

[3] A. Cerpa, J. Elson, et al. Habitat Monitoring: Application Driver for Wireless Communication Technology. In *ACM SIGCOMM Workshop on Data Communications*, Apr. 2001.

[4] R. J. Clark, M. Ammar, and K. Calvert. Multi-Protocol Architecture as a Paradigm for Achieving Inter-Operability. In *Proceedings of IEEE INFOCOM*, Apr. 1993.

[5] The Familiar Project. `http://familiar.handhelds.org/`.

[6] J. Heidemann, F. Silva, C. Intanagonwiwat, et al. Building Efficient Wireless Sensor Networks with Low-Level Naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

[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 T. Imielinski and H. Korth, editors, *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] J. Kahn, R. Katz, and K. Pister. Next Century Challenges: Mobile Networking for "Smart Dust". In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking (MOBICOM '99)*, Aug. 1999.

[12] 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.

[13] S.-F. Li, R. Sutton, and J. Rabaey. Low Power Operating System for Heterogeneous Wireless Communication Systems. In *PACT 01 Conference*, Sept. 2001.

[14] Microsoft Corp. Microsoft Windows CE. `http://www.microsoft.com/windowsce/embedded/`.

[15] Palm Corp. Palm OS 5 Development Overview. `http://www.palmos.com/dev/support/docs/palmos50/`.

[16] 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.

[17] G. Pottie and W. Kaiser. Wireless Integrated Network Sensors. *Communications of the ACM*, 43(5):51–58, May 2000.

[18] An Architecture Overview of QNX. `http://www.qnx.com/literature/whitepapers/archoverview.html`.

[19] A. Sinha and A. P. Chandrakasan. Operating System and Algorithmic Techniques for Energy Scalable Wireless Sensor Networks. In *2nd International Conference on Mobile Data Management (MDM 01)*, Jan. 2001.

[20] D. L. Tennenhouse and D. J. Wetherall. Towards An Active Network Architecture. *Computer Communication Review*, 26(2), Apr. 1996.

[21] VxWorks 5.4 Datasheet. `http://www.windriver.com/products/html/vxwks54_ds.html`.

[22] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH*, Apr. 1998.