Active Message Communication for Tiny Networked Sensors

Philip Buonadonna, Jason Hill, David Culler

Abstract—

We present an implementation and evaluation of an Active Messages based communication system for tiny, wireless, networked sensors. The implementation includes two major software components. The first is the device based operating program which includes the communication subsystem, dispatch loop and AM handlers. The second is a communication library for general purpose host computers. Using an Atmel 8535 based device and an Intel Pentium II PC, we demonstrate an ad hoc networking application that uses Active Message primitives for multi-hop route discovery and packet delivery on silver dollar sized devices. We also make observations about the applicability of TCP/IP to the Tiny Networked Sensor regime.

I. Introduction

The Post-PC era of computing has introduced an array of small devices that perform a variety of specific functions. Cellular phones, pagers and portable digital assistants are common examples of these. As technology progresses, however, devices will continue to become smaller and more specialized. One class of device that is beginning to emerge is the tiny networked sensor. These machines are characterized by an embedded processor capable of a couple of MIPS, a limited amount of storage (i.e. 4Kb or less), a small power source, a short range radio, and an array of sensors and/or actuators. The practical applications of such mini-devices range from environmental monitoring to micro-robots capable of performing microscopic scale tasks. While the functionality of an individual device is limited, a collection of them working together could accomplish a range of high-level tasks.

Paramount to achieving the goal of cooperative mini-devices is the design of the communication subsystem. The demands here are numerous. It must be efficient in terms of memory, processor, and power requirements so that it falls within the constraints of the hardware. It must also be agile enough to allow multiple applications to simultaneously use communication resources.

Such demands inhibit the use of legacy communication systems (e.g. TCP/IP) which have seen a great deal of success in more conventional settings. The low level networking protocols of legacy stacks typically require many kilobytes of memory at a minimum and their performance is dependent on a fast processing component. Routing protocols implemented above these (e.g., OSPF) are complex and place bandwidth demands that become untenable for links with kilobits per second throughput. Finally, common high level software abstractions for using the network, such as sockets, are not well suited to the constrained environment of tiny network devices. Interfaces centered on "stop and wait" semantics do not meet the power and agility requirements of these small devices.

In this paper, we investigate the application of the Active Messages paradigm to mini-device communication. We believe that there is a fundamental fit between the event based nature of network sensor applications and the event based primitives of the Active Messages communication model. It is a framework that handles the limitations of these devices, yet provides a powerful programming environment capable of supporting a wide space of concurrent applications.

Active Messages is centered on the concept of integrating communication and computation, as well as matching communication primitives to hardware capabilities. Tiny networked devices must take advantage of the efficiencies that can be achieved by this matching. Furthermore, the inability to support a large number of simultaneous execution contexts requires that computation and communication be overlapped so that valuable computational resources are not wasted.

We demonstrate an implementation of Active Messages for a prototype networked sensor mini-device with a 4 MHz Atmel AVR 8535 Microcontroller, 512 Bytes of RAM and 8KB of FLASH memory. We show an ad-hoc networking application built on top of the Active Message primitives that performs automatic topology discovery and data collection from autonomous nodes.

In the next section, we outline the fundamental aspects of networked mini-devices that place strong requirements on the communication model. Section 3 provides background information for Active Messages, autonomous sensor devices and their operating system architecture. Sections 4 and 5 present a high-level overview and low-level details of our tiny Active Messages implementation respectively. Section 6 describes a demonstration application built using tiny Active Messages. Section 7 evalu-

ates our implementation in terms of usability and raw performance. Section 8 presents retrospectives. Section 9 presents work related to network sensors. We conclude with options for future work.

II. Requirements

Networked sensors represent a new design paradigm that is being enabled by significant advances in MEMS structures and low power technology. Their communication requirements are determined both by their physical design characteristics and their intended use scenarios.

The ability to create sensors and actuators with IC technology and integrate them with computational logic has created an abundance of low-power sensors. Combining these new sensors with the extensive work being done to develop low power wireless communication [5] provides the basis for networked sensors. They generally include a tiny microcontroller, environmental detectors, and a low power radio. This integration of computation, communication, and physical interaction together in silicon has created the unique opportunity to shrink these devices down to microscopic scales. Never before has it been so predictable that a physical device will track Moore's law down in physical size.

This trend toward highly integrated systems places strict limits on physical capabilities of the devices. The desire to continually shrink these devices, makes it necessary to be as efficient as possible. Specifically, modern devices must operate in only kilobytes of program memory and hundreds of bytes of RAM.

Additionally, their interaction with the physical world places real time constraints on their operation. If they are receiving a communication over the radio, missing a deadline by a fraction of a second will cause the transmission to be lost. Additionally, these sensors will be engaged in multiple simultaneous interactions with the physical world – sensing, communication, routing – which forces them to be agile. It is clear that they must be able to support intensively concurrent operations. This necessity makes the programming paradigm of busy-waiting until data is available impractical.

These constraints are compounded by the fact that the central processor in the microsensor is directly connected to the I/O devices. Traditional systems generally use coprocessors and dedicated hardware to compose a system hierarchy. This allows for a high degree of physical parallelism, which simplifies the task of the central processor. However, high levels of physical parallelism are not practical for network sensors. In our system, the central processor must service every bit of a radio transmission individually, while simultaneously tending to the collection of data from sensors and the execution of application level code. Keep in mind this must be done by a processor capable of only two MIPS.

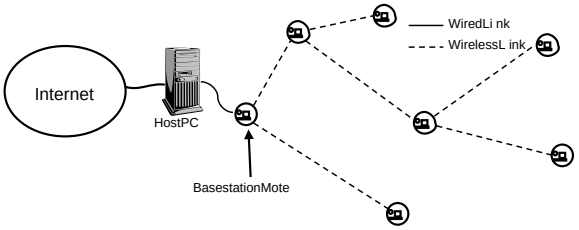


Fig. 1
A typical sensor network physical topology. The Host PC forms the device's logical connection to external applications. The device attached to the PC forms the basestation.

The intended usage scenarios for networked sensors also dictate additional requirements for the communication system. In general, the user will want to be able to quickly deploy a large number of low cost devices without having to configure or manage them. This means that they must be capable of assembling themselves into an ad hoc network. The mobility of individual sensors and the presence of RF interference means that the network will have to be capable of reconfiguring itself in a matter of seconds. Figure 1 shows a typical network topology.

Additionally, these devices must rely on small batteries or ambient energy for power and run for long periods of time without maintenance. To conserve power, it is important that a communication system is capable of multi-hop routing. It is more efficient to relay data multiple times towards a destination than to increase the transmission strength so that it can be received by the endpoint directly. Efficient use of the CPU can also lead to power savings. While Moore's Law will increase the computational abilities for a given size device, a 20% increase in efficiency will always translate into a 20% increase in battery life. Software efficiencies can be translated into power savings by reduction of clock speeds and CPU voltages.

Finally, these sensors will typically be tailored to a specific task. This tailoring will take the form of specialized software routines as well as customized hardware. This diversity in design and usage puts extra requirements on the overall system design. Specifically, it forces the system to be architected in a highly modular fashion that allows mul-

tiple application components to coexist. This precludes any single application from assuming that it can utilize all of the available CPU resources. Clearly, any communication model that busy-waits inherently assumes that no other applications are running on the device and is impractical for this regime.

III. Building Blocks

Instead of scaling PC based communications down, we believe that it is possible to use the emergence of this new design space as an opportunity to re-evaluate current architectures with respect to modern technological advancements. We have used the Active Messages communication model and the event based TinyOS as building blocks for our solution.

A. Active Messages

Active Messages (AM) is a simple, extensible paradigm for message-based communication widely used in parallel and distributed computing systems [14], [21]. Each Active Message contains the name of a user-level handler to be invoked on a target node upon arrival and a data payload to pass in as arguments. The handler function serves the dual purpose of extracting the message from the network and either integrating the data into the computation or sending a response message. The network is modeled as a pipeline with minimal buffering for messages. This eliminates many of the buffering difficulties faced by communication schemes that use blocking protocols or special send/receive buffers. To prevent network congestion and ensure adequate performance, message handlers must be able to execute quickly and asynchronously.

Although Active Messages has its roots in large parallel processing machines and computing clusters, the same basic concepts can be used to the meet the constraints of networked mini-devices. Specifically, the lightweight architecture of Active Messages can be leveraged to balance the need for an extensible communication framework while maintaining efficiency and agility. More importantly, the event based handler invocation model allows application developers to avoid busy-waiting for data to arrive and allows the system to overlap communication with other activities such as interacting with sensors or executing other applications. It is this event centric nature of Active Messages which makes it a natural fit for these devices.

B. Pure Event-based Programming

Choosing to harness the power of an event based operating system will clearly have a significant impact on the communication paradigm. To construct our communication system, we use the event based TinyOS [9]. This operating system is designed to meet the critical needs of the networked sensor design regime. Its execution model is similar to an FSM based model, but considerably more programmable. It is designed to support the concurrency intensive nature of networked sensors while allowing for efficient modularity. Specifically, TinyOS's event model allows for high concurrency to be handled in a very small amount of space. A stack-based threaded approach would require orders of magnitude more memory than we expect to have available. Furthermore, the Active Messages event based communication abstraction fits well inside the TinyOS event based programming model.

The TinyOS design is centered on a tiny scheduler and a graph of *components*. Components interact by receiving commands from "higher level" components and handling events from "lower level" components. To facilitate modularity, each component declares the commands and events it uses and handles.

Event handlers are invoked to deal with hardware events, either directly or indirectly. The lowest level components have handlers connected directly to hardware interrupts, which may be external interrupts, timer events, or counter events. Events propagate up through the component hierarchy as necessary. In order to perform long-running computation, components can request to have tasks executed on their behalf. Once executed by the scheduler, these tasks run to completion and execute autonomously with respect to other tasks. However, they can be periodically interrupted by higher priority events. Because a task must complete before a subsequent task can be executed, they must never block or busy-wait. Commands, tasks, and event handlers all have access to a component's private, persistent state. This model allows simple components to be composed together into complex applications.

IV. Tiny Active Messages

In bringing Active Messages out of the high performance parallel computing world and down into this low power design regime, we have attempted to preserve the basic concepts of integrating communication with computation and matching communication primitives to hardware capabilities. The basic paradigm of typed messages causing handlers to be invoked upon arrival matches up well with the event based programming model supported by TinyOS and demanded by the underlying sensor hardware. The low overhead associated with event based notification is complementary to the limited resources of networked sensors. Applications do not need to waste resources while waiting for messages to arrive. Additionally, the overlap of computational work with application level communication is essential. Execution contexts and

stack space must never be wasted because applications are blocked, waiting for communication. Essentially, the active messages communication model can be viewed as a distributed eventing model where networked nodes send each other events. While quite basic, we believe that all applications can be built on top of this primitive model.

In order to make the active messages communication model a reality, certain primitives must be provided by the system. We believe that the three basic primitives are: best effort message transmission, addressing, and dispatch. More demanding applications may need to build more functionality on top of these primitives, but that is left for the applications developer to decide. By creating the minimal kernel of a communication system, all applications will be able to build on top of it.

Additionally, it is likely that there will be a large variety of devices with different physical communication capabilities and needs. By building the communication kernel as three separate components using the TinyOS component model, developers can pick and choose which implementations of the basic components they need. This can take the form of selecting from a collection of delivery components that perform different levels of error correction and detection. However, by providing a consistent interface to communication primitives, application developers can easily transfer their applications to different hardware platforms.

Just as there will be various implementations of the core components for the developer to choose from, various other extensions will be available such as reliable delivery. This is similar to the design of Horus [20], which attempted to have modular PC based communication protocols where application developers could chose from a variety of building blocks including encryption, flow control, and packet fragmentation. It is extremely advantageous to be able to customize protocols when dealing with Networked Sensors due to their extreme performance constraints and their large diversity of physical hardware.

Finally, the selection of an event based communication mechanism does not preclude the use of a threaded, blocking, execution model. An event-based model can easily be transformed into a threaded model through the use of a queue, where an event simply places the data into a queue structure that can be accessed by the thread. When the queue is empty, the thread can block until data arrives. On the contrary, it is difficult to switch from a threaded implementation to an event-based model. Similarly, the immediate propagation of messages to the application layer does not prevent the use of buffers to temporarily hold messages until the application is ready to deal with them. An application level buffer component could be used to accomplish this. However, including extra buffers inside the communication primitives precludes the application from eliminating them.

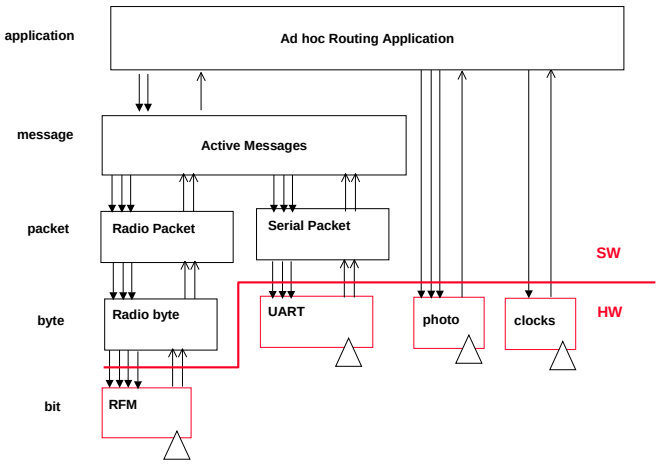


Fig. 2
AD-HOC NETWORKING APPLICATION COMPONENT GRAPH.

V. IMPLEMENTATION

In this section, we present the design of our Tiny Active Messages implementation. We believe that the basic primitives that we have provided are all that is needed to construct application level protocols the meet any application's needs. The discussion includes the device and host PC components and aspects that are common to both. We will follow with an evaluation section that demonstrates how we have successfully used these primitives to construct an ad-hoc networking application based on data collection using networked sensors.

A. Components

The interface to our messaging component is quite simple. It accepts TinyOS commands from the applications to initiate message transfers and fires off events to message handlers based on the type of messages that have been received. There is an additional event that signals the completion of a transmission. Send commands include the destination address, handler ID, and message body. Internally, our Active Messages component performs address checking and dispatch and relies on sub components for basic packet transmission. Figure 2 shows the complete TinyOS component graph of or application.

The underlying packet level processing components simply perform the function of transmitting the block of bytes out over the radio. We assume that this is a best effort transmission mechanism. While we do not expect reliable, error free delivery, we do assume that there will be

some basic logic that attempts to avoid transmission collisions. The interface to the packet level component provides a mechanism to initiate the transmission of a fixed size, 30 byte packet, as well as two events that are fired when a transmission or a reception are complete. We have multiple implementations of packet level components each providing different levels of error correction or detection. They include, basic transmission without any error detection or correction, CRC checked packets that have error detection, and forward error corrected packets that provide basic error correction as well as error detection.

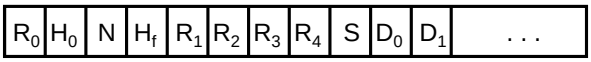
Additionally, we have implemented a Host PC package that consists of a software library that is linked into applications. The library communicates over the PC serial port to a special base station sensor that has both a RS232 communication channel as well as RF communication. This provides a way to quickly develop communication bridges that bring data collected and transmitted on the networked sensors into a more traditional computing environment. Using this library, it is simple to create a bridge that retransmits messages collected from the networked sensors onto the Internet using TCP/IP or UDP.

B. Packet Format

The first two bytes of a received packet are used to identify the destination of the packet (R_0) and the ID of the message handler that is to be invoked on the packet (H_0). The AM component first checks that the address matches the local address and then it invokes the listed handler, passing on the remaining 28 bytes of the packet. In the event that the message is bound for a handler that is not present on the receiving device, the packet is ignored. The dispatch routine that is used by the message handler is automatically generated at compile time based on the message handlers that are present. This is done to eliminate the need for expensive handler registration mechanisms.

C. Multi-hop Packet Format

In order to demonstrate how application specific needs could be met by building on top of the basic Active Messages primitives, we have developed a component that supports source based multi-hop routing. To accomplish this, we have defined a generic message format and specialized routing handler. This format, depicted in Figure 3, dedicates seven additional bytes to allow a maximum of 4-hop communication. Four of these bytes are used to hold the intermediate hops of the route (R_1,R_2,R_3,R_4), one is used for the number of hops left (N), one is used to store the source of the packet (S), and one is used for the handler ID that is to be invoked once the message arrives at its destination (H_F). In this instance, the multi-hop router



R_0 - Next Hop
H_0 - Next Handler
N - Number of Hops
H_0 - Destination Handler
R_1, R_2 ,R_3 ,R_4 - Route Hops
S - Sending Node
D_0, D_1 ... - Payload

Fig. 3
MULTI-HOP PACKET FORMAT.

is simply the handler of a typed message. More complex routing information is stored inside the message and used by application level handlers to route the packet to its next destination. This simple component can be used in any application that wishes to have source based routing. Additionally, applications that need other functionality can seamlessly coexist along side of it.

While the packet is in-route, H_0 is set to zero: the routing handler. In response to the reception of a packet, the routing handler decrements the hop count and rotates in the next hop and pushes the local node address to the end of the route chain. This process records the route that the packet has taken in the route table so that the recipient knows how to route a response packet. If the next hop is the final destination (number of hops is one), the routing handler inserts the destination handler, H_F, into H_0.

D. Special Addresses

In developing sample applications to test the usability of our messaging layer, two special addresses were defined. The first special address that was needed was the broadcast address. The concept of a one-to-all broadcast greatly simplifies the route discovery and exploration algorithms. Combining this with routing handlers designed to record the path that a packet has taken yields a trivial implementation of a route discovery application. In its simplest form, an application can send a two-hop packet to the broadcast address followed by its own address. This will cause any device that is in range to respond with its own address recorded in the packet that the original device receives.

Secondly, a special address was chosen for the Host PC in the device virtual network. Arbitrarily chosen to be 0x7e, a device receiving a packet for this destination forwards the packet to the local data UART instead of the radio. This exposes the basic need to have the notion of gateway addresses that get treated specially.

VI. Demonstration

We have built an ad hoc networking application to collect information from a set of nodes that have been randomly distributed throughout the environment. The application uses the Active Message primitives to explore routing topology and then to propagate information towards a central collection node. Additionally, as sensors are relocated, the application automatically reconfigures itself in light of the new routing topology. We have selected this application because we believe that this closely mirrors real world networked sensor applications.

To construct our ad-hoc network we use a variation on a Destination-Sequenced Distance Vector [18] algorithm tailored to having all nodes transmit to a basestation. While not the most efficient algorithm, it is straightforward to implement and understand. It servers the purposes of demonstrating the capabilities of the system. Our framework can easily be used to implement and evaluate more sophisticated networking algorithms.

The system starts out with zero knowledge of the identity or topology of sensors that are present. Each mobile node knows only its own identity. Additionally, the base station knows that it is directly connected to the host PC. The base station is the origin of all routing update messages. It periodically broadcasts out its identity and that it has connectivity to the host PC. Devices in direct communication range of this base station receive the message and use it to update routing information. They then rebroadcast a new routing update to any devices in range that there is a path to the base station through them. Devices remember the first routing update that they hear, which corresponds to the shortest path to the base station. In order to prevent cycles in the routing topology, time is divided into eras and route updates are broadcast once per era. Without these boundaries, nodes may get confused as to which route updates they should latch onto.

For information collection, each node periodically generates and transmits information, and it participates in the routing of network data towards the base station. A device's transmissions are addressed to the device ID that was received in the last routing update. The recipient will repeat the same process until the packet reaches the base station. In this system, each node simply knows the identity of the next hop that will bring the packet closer to the base station.

In addition to forwarding the packet, the identities of the intermediate hops are stored in the packet as it travels to the base station. This allows us to determine the overall routing topology that the network is using. By looking at the route traveled by several packets from different sources, we can eventually determine the topology of the entire system. For demonstration purposes, we feed this information into a display application that plots the current routing topology graphically. This takes the form of a tree rooted at the base station node with edges corresponding to communication routes. One example is shown in Figure 4. Each node is labeled with the ID of the device and decorated with a box that represents light level sensor reading being collected. Information is also displayed showing the number of data packets that have traveled through a node and received from a node.

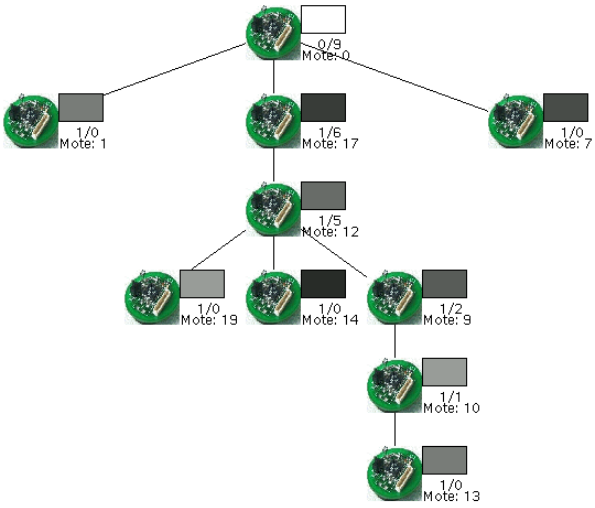


Fig. 4
Routing diagram generated by analysis of messages collected at Host PC.

VII. Evaluation

While it is important to verify that our communication mechanisms perform well, the quantitative metrics are not as important as the qualitative metrics associated with programmability. Thus, we have developed a sample application that demonstrates the power of the primitives that we have included. For completeness, we will also provide a small number of performance benchmark results.

A. Utility of Active Messages

In the routing application, there are two types of active messages. One message type is dedicated to route update messages, and the other is dedicated to data collection messages. Surprisingly, we were able to implement both the base station and remote node applications in approximately 100 lines of commented code (approx. 70 lines of actual C statements).

The route update message handler performs the func-

tion of recording the received information in the routing table and then initiating the retransmission of the propagated route update message. Similarly, the data collection message handler responds to the receipt of a packet that needs to be forwarded towards the base station. This handler checks the routing table, updates the payload of the packet to record that it transitioned through the local node and then sends the packet towards the recipient stored in the routing table.

Finally, there is a third type of event that is handled by the application. There is a clock event that periodically triggers the node to begin collection of data from its sensors and then transmits the newly collected data towards the base station. These three event handlers are all that are needed to pull off this seemingly complex application.

In comparison, there are several ways that this application could be mapped into a threaded execution model. One method would involve creating separate application level threads to be in charge of local data collection and network management. The networking thread would block, waiting for data to be received. Upon arrival, it would either forward the data to the next node or use the information to update the local routing table. Meanwhile, the data collection thread would be periodically collecting sensor readings and sending them out on the network. Inherently, this model would involve significantly more overhead on the part of the operating system managing these execution contexts. We have yet to mention the additional execution contexts that must support the threaded model. Furthermore, this increased complexity would not be offset by a simplified programming experience. A majority of the application level code would remain unchanged. In fact, it is likely that the code would be more complex than the Active Messages based implementation because the application developer would be responsible for creating multiple threads and performing dispatch on messages received over the network.

B. Raw Metrics

Here we attempt to quantitatively evaluate our implementation with respect to the traditional metrics of round trip message time and throughput. For networked sensors these do not completely reflect the impact of the communication subsystem on all the design parameters of concern. Thus we include two additional measurements that are critical for small devices: software footprint and energy usage.

The round trip time (RTT) metric measures the time for a message to be sent from the host PC to a specific sensor device and back. Figure 5 presents the RTT results for various route lengths through the network. A route length

Fig. 5

ROUND TRIP TIMES FOR VARIOUS ROUTE LENGTHS. NOTE THAT ONE HOP MEASURES THE TIME FOR A MESSAGE BETWEEN THE HOST PC AND BASE STATION DEVICE.

of one measures the Host-PC to basestation RTT (40ms) and reflects the cost of the wired link, device processing, and the host OS overhead. The difference between request arrival and reply transmission of .3 ms shows that the Active Message layer only accounts for .75% of the total RTT time over the wired link. This decreases when compared to the longer transmission times of the wireless link.

Component	Cumulative Time (msec)
First bit of request on device	0
Last bit of request on device	15.6
First bit of reply from device	15.9
Last bit of reply from device	32.8
First bit of next request on device	40.0

TABLE I

CUMULATIVE TIME PROFILE FOR A SINGLE HOP RTT TEST.

Table 1 presents a cumulative profile of the single hop RTT. For routes greater than one hop, the RTT also includes the latency of the wireless link between two devices. The difference between the two and one hop RTT yields the device-to-device RTT of 78ms. The additional cost of the device-to-device message is accounted to the slower link rate and the encoding scheme. At 10Kbps and 50% encoding overhead, the one way link transmit time is approximately 36ms.

The wireless link and encoding schemes also determine the maximum throughput of the communication system. Using the 4b6 encoding algorithm that is required by the RFM radio [2] being used, the maximum realizable throughput is 833 bytes/sec.

The software footprint refers to the total number of bytes occupied by a software component on the device

memory resources. The Active Message layer occupies a total of 322 bytes. The total device binary is 2.6 Kbytes and includes the packet level, byte level and bit level controllers, the AM component and the routing application. Of the Active Message footprint, 40 bytes are used for static data. This includes a 30 byte buffer, a one-byte local address, 4 bytes of state and 5 bytes for compiler alignment.

The last, and perhaps most critical, metric is power consumption. The event model supported by Active Messages enables the device to enter an idle state when no communication or computation is being performed. We measured the power consumption for this idle state, the peak power consumption and the energy required to transmit one bit. The results are presented in Table 2.

Idle State	5 μAmps
Peak	5 mAmps
Energy per bit	1 μJoule

TABLE II
POWER AND ENERGY CONSUMPTION MEASUREMENTS.

VIII. RETROSPECTIVE

The Active Message model exploits the symmetry between the networked sensor hardware and event based communication mechanisms. The physical structure of the network sensor that we have described forces the developer into an event-based model. This is in part due to the large number of devices that a single processing unit must deal with. However, it is also driven by the fact that energy is the most precious resource. While the CPU may have time to poll across multiple devices or internal queues, it cannot afford the energy that this would consume. The event-based model that we have developed eliminates all polling from the architecture. When all tasks have completed, the CPU goes to sleep. It only awakes when a hardware event triggers it. This is identical to the behavior of an application written above the active messages layer. When it completes its work, it terminates until an active messages event awakes it. Furthermore, in both cases the nature of the event tells it exactly what is to be done. The active messages paradigm is a reflection of the underlying hardware properties.

Interestingly, many Active message implementations on large parallel and distributed systems do not use events. Studies have shown that polling in these large scale systems to achieve a higher level of performance due the high computational overhead of interrupts and events [12]. Essentially, they are forced to trade CPU cycles for improvements in latency or bandwidth figures. For networked sensors, power limitations make these types of schemes impractical. This is because the underlying operating systems prevent the use of user level interrupt handlers. Interrupts must be handled by the kernel and indirectly relayed to the application. However, starting over on this new class of device has allowed us to redesign the underlying operating system.

Another advantage to using events is that polling based I/O mechanisms see significant performance degradation when the number of interfaces that must be periodically checked increases. Performance measurements using the select system call to poll across multiple open sockets show that as the number of sockets increases the performance decreases [16]. Essentially, the solution is to use a single event queue to limit the number of places that need to be checked in determining which sockets have data ready. The same approach of reducing the number of places that need to be polled to improve performance has been used in high performance communication implementations. We have taken this idea a step further: we have reduced it to zero things to poll. One of the main reasons that high performance systems have not taken the same step is because traditional Unix kernels are not designed to support this mode of operation. More specifically, interrupts are expensive on modern systems and, it is not easily possible to have an interrupt handler directly invoke a user level thread.

It is intuitive to believe that communication models that have been successful in the PC world could be applied to this new design regime. However, we believe that there are fundamental differences that inhibit the use of established schemes. Specifically, we argue that a traditional socket based TCP/IP communication model is not optimal for the networked sensor regime. While sockets have seen a great deal of success in modern PCs, it is not clear that they can be efficiently mapped down into low-power, wireless systems.

First, the use of a traditional socket based abstraction as a communication mechanism forces the systems into a thread based programming model. This is because sockets have a stream-based interface where the user application polls or blocks as it waits for data to arrive. As argued in [9], it is not currently possible to support the use of thread based programming models on the class of device envisioned for the tiny networked sensor because of overhead associated with context switches and the storage of inactive execution contexts. While the wire line protocols of UDP and TCP do not force a programming model on the user, all popular implementations do.

It is also important to consider the “bits on the wire”

overhead. Communication is extremely expensive for network sensors; while transmitting, a radio will consume more power than the CPU. This makes it very advantageous to transmit as few bits as possible. Assuming that a packet based communication mechanism will be used, this translates into wanting the fewest number of overhead bits per packet. In TCP/IP and UDP, these bits take the form of sequence numbers, addresses, port numbers, protocol types, etc. In all, a TCP/IP packet has an overhead of 48 bytes. This is not counting the overhead associated with acknowledgements and retransmissions. While these packets are useful for a general class of applications, they should not be forced on all applications.

Third, there is a significant amount of overhead inherent in the TCP/IP protocol that makes it ill-suited to this class of device. This includes the memory management associated with a stream based interface. The networking stack must buffer incoming data until the application requests it whereupon it must be copied into the application's buffer while any remaining data remains buffered by the protocol stack. This buffer management greatly increases complexity and overhead. If there is insufficient buffer space, data will be discarded without informing the application. The stream based communication model also has significant overhead on the sender side in the form of intermediate copies and data fragmentation. While advanced implementations have attempted to perform zero copy TCP/IP [17], these require significant increases in complexity.

There is an assortment of Operating Systems that provide TCP/IP based network connectivity to embedded devices. These systems include many commercial real time operating systems such as VxWorks [4], OS-9 [15], PalmOS [1] and QNX [11], [13]. However, these systems have been designed to operate on hardware equivalent of a 90's PC. While this allows them to handle the complexity and overhead of the BSD sockets based TCP/IP communication model, it is no surprise that these real time operating systems consume significantly more resources than are currently available on the class of hardware that we are targeting. In general, memory footprints and CPU requirements are orders of magnitude beyond the hardware capabilities [9].

Moore's law states that it will be possible to place modern microkernels on tiny networked devices within a few years. However, it is envisioned that hardware of tiny networked devices will exploit technological innovation to achieve unprecedented form factors opening new frontiers to computer science. Tiny networked sensors will follow technological trends towards the microscopic physical size while maintaining a constant level of performance. This is in contrast to traditional paradigms of increasing the capabilities for a given physical size.

Efforts have been made to create reduced complexity implementations of TCP/IP such as TinyTCP [7]. They have alleviated some of the protocol overhead by creating a TCP implementation without a socket-based interface. However, in the case of TinyTCP, the programming model does not allow multiple applications to coexist. Its "busy waiting" style of programming completely consumes the processor during communication. This is clearly not acceptable for the concurrency intensive operation of network sensors. Moreover, the limited implementations of TCP/IP are centered on achieving minimal connectivity for socket endpoints. They cannot handle application scenarios where individual nodes act as forwarding gateways for other nodes. For example, the Seiko iChip [3] provides an extremely low power TCP/IP stack in hardware, but was designed for use in client devices that create a small number of connections. While there is a large class of devices that fit into that design paradigm, the networked sensor is not one of them. Particularly, multi-hop routing applications need the ability to have low power intermediated nodes forward data on behalf of other nodes.

IX. Related Work

On the hardware side, the Smart Dust Project [19] is developing a millimeter cubed integrated network sensor. There is also work in developing low power hardware to support the streaming of sensor readings over wireless communication channels [5]. However, both of these systems have focused on the enabling hardware rather than on the programming interface that will be used by application developers.

The Wireless Application Protocol (WAP) [10] addresses many of the same wireless device issues presented in this paper (e.g. power and CPU constraints). It's standards cover a vertical section of the network protocol stack from application interfaces to low-level transports. However, WAP is targeted mainly at client-server type applications where the wireless device presents a human interface. Our work investigates a design space of small autonomous devices that that may operate in large (i.e. hundreds or thousands) collectives.

Additionally, extensive work is being done to explore applications enabled by network sensors. Piconet [6] and The Active Badge Location System [22] have explored the utility of networked sensors. Work at ISI [8] has explored routing alternatives and power saving mechanisms for the networked sensor regime. These systems have focused on application level optimizations, assuming the existence of a basic messaging protocol, such as Tiny Active Messages.

X. Future Work

It is clear that the next step is to demonstrate our architecture supporting more complex ad hoc routing protocols. We have started with an algorithm that is simple to reason about but is not optimized for efficiency. We need to harness the significant research has gone into the development of more advanced algorithms.

Additionally, while our system does currently support the ability to broadcast messages out to everyone, another primitive that we believe should be added is the support of a multicast like mechanism. This could be used to disseminate information to a collection of devices that are working in concert. An example would be to the dissemination of configuration information. The goal is to prevent retransmission of messages whenever possible. This would be a simple single hop optimization not designed to solve the routing issues associated with multicast subscription. However, more traditional versions of multicast routing layers would be enabled by it.

Finally, this could be used to setup virtual channels or virtual networks in a sensor net. In addition to receiving messages bound for multicast addresses, devices could be configured to use a separate dispatching mechanism for these messages. This could be another way of providing per application handler naming mechanisms. For example, applications could specify that their special handlers be invoked for messages that arrive on certain multicast channels. This could be used to selectively transmit information to a collection of devices deployed in a heterogeneous environment.

XI. Conclusion

The emergence of networked sensors and actuators has created a wide space of new problems in distributed systems design. One of the driving forces behind these new problems is a shift in the perception of performance: high-endurance, low energy use, and modularity over FLOPS and throughput. In this paper, we have investigated a communication architecture for networked sensors based on the Active Messages model. We have built and analyzed a reference implementation of Active Messages for a prototype wireless networked sensor which include components for Tiny OS and a library for a host PC. This implementation demonstrates the ability to handle the concurrency intensive demands of the networks sensor regime. Additionally, the communication layer consumes 322 bytes of memory and consumes less than 5μAmps while in the idle state. A ad-hoc networking application built on top of the implementation demonstrates the extensibility of the Active Messages paradigm to specific applications.

Admittedly, there is one potential drawback of using Active Messages: It's not TCP/IP. The recent ubiquitous deployment of IP has thrust it to be the protocol of choice for many network applications. However the power and scalability demands of the networked mini-device forces a rethinking of network protocols. We have presented several reasons why socket based, TCP/IP, communication paradigms are not appropriate for tiny networked devices. In contrast, we have shown how the Active Message paradigm is a natural fit for the demands of these devices; it provides an efficient, agile, and extensible communication mechanism.

References

[1] PalmOS Software 3.5 Overview. `http://www.palm.com/devzone/docs/palmos35.html`.

[2] RF Monolithics. `http://www.rfm.com/products/data/tr1000.pdf`.

[3] Seiko iChip. `http://www.seiko-usa-ecd.com/intcir/html/assp/s7600a.html`.

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

[5] B. Atwood, B.Warneke, and K.S.J. Pister. Preliminary circuits for smart dust. In *Proceedings of the 2000 Southwest Symposium on Mixed-Signal Design*, San Diego, California, February 27-29 2000.

[6] F. Bennett, D. Clarke, J. Evans, A. Hopper, A. Jones, and D. Leask. Piconet: Embedded mobile networking, 1997.

[7] Geoffrey H. Cooper. Tiny TCP/IP. `http://www.csonline.net/bpaddock/tinytcp/default.htm`.

[8] Deborah Estrin, Ramesh Govindan, and John Heidemann. Scalable coordination in sensor networks.

[9] Jason Hill et. al. System Architecture Directions for Networked Sensors. `http://www.cs.berkeley.edu/~jhill/papers/tos.pdf`.

[10] WAP Forum. Wireless application protocol white paper, October 1999.

[11] Dan Hildebrand. An Architectural Overview of QNX. `http://www.qnx.com/literature/whitepapers/archoverview.html`.

[12] Lok T Liu and David E Culler. Measurements of active messages performance on the cm-5. Technical Report UCB//CSD-94-807, University of California at Berkeley, Department of Computer Science, May 94.

[13] QNX Software Systems Ltd. QNX Neutrino Realtime OS . `http://www.qnx.com/products/os/neutrino.html`.

[14] Alan M. Mainwaring and David E. Culler. Design challenges of virtual networks: Fast, general-purpose communication. In *Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practise of Parallel Programming (PPoPP'99)*, volume 34.8 of *ACM Sigplan Notices*, pages 119–130, A.Y., May 1999.

[15] Microware. Microware OS-9. `http://www.microware.com/ProductsServices/Technologies/os-91.html`.

[16] Peter Druschel Mohit Aron and Willy Zwaenepoel. A scalable and explicit event delivery mechanism for UNIX. In *Proceeding of the USENIX 1999 Annual Technical Conference*, June 1999.

[17] Michel Muller. *Zero-Copy TCP/IP with Gigabit Ethernet*. PhD thesis, Institute for Computer Systems, ETH Zurich, 1999.

[18] Charles E. Perkins and Pravin Bhagwat. . highly dynamic destination-sequenced distance vector routing for mobile computers. In *In Proceedings of the SIGCOM '94 Conference on Communications Architectures, Protocols, and Applications*, August 1994.

[19] K. S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes, 1999.

[20] R. Van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in horus. In *In Proceedings of the ACM Symposium on Principles of Distributed Computing*, August 1995.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Qld., Australia, May 1992.

[22] R. Want and A. Hopper. Active badges and personal interactive computing objects, 1992.