A Network-Centric Approach to Embedded Software for Tiny Devices

David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo

University of California at Berkeley, Intel Research at Berkeley, Berkeley CA 94720, USA

Abstract The ability to incorporate low-power, wireless communication into embedded devices gives rise to a new genre of embedded software that is distributed, dynamic, and adaptive. This paper describes the network-centric approach to designing software for highly constrained devices embodied in TinyOS. It develops a tiny Active Message communication model and shows how it is used to build non-blocking applications and higher level networking capabilities, such as multihop ad hoc routing. It shows how the TinyOS event-driven approach is used to tackle challenges in implementing the communication model with very limited storage and the radio channel modulated directly in software in an energy efficient manner. The open, component-based design allows many novel relationships between system and application. [1]

1 Introduction

The emergence of compact, low-power wireless communication, sensors, and actuators in the technology that supports the ongoing miniaturization of processing and storage is giving rise to entirely new kinds of embedded systems and a fundamentally new genre of embedded software. Historically, embedded systems have been highly engineered to a particular task. For example, a disk drive controller sits between a standardized command/response channel and the disk head assembly, with its rotation sensors, positioning actuators, and read/write heads. The controller software is a highly orchestrated command processing loop to parse the request, move the heads, transfer the data, perform signal processing on it, and respond. The system is sized and powered for the particular application. The software is developed incrementally over generations of products and loaded into a device for its lifetime. An engine ignition controller is even more specialized to performing a particular sense/actuate loop autonomously.

The new kinds of embedded systems are distributed, deployed in environments where they may not be designed into a particular control path, and often very dynamic. The fundamental change is communication; collections of devices can communicate to achieve higher level coordinated behavior. For example, wireless sensor nodes may be deposited in offices and corridors throughout

[1] This research was supported by the Defense Advanced Research Projects Agency, the National Science Foundation and Intel Corporation

a building, providing light, temperature, and activity measurements. Wireless nodes may be attached to circuits or appliances to sense current or to control usage. Together they form a dynamic, multihop routing network that connects each node to more powerful networks and processing resources. Through analysis of radio signal strength and other sensory nodes, nodes determine their location. They tap into local energy sources, perhaps using photovoltaic cells or nearby telephone or AC lines, to restore their energy reserves. Nodes come and go, move around, and are affected by changes in their environment. Collectively, they adapt to these changes, perform analysis of usage patterns and control lighting, temperature, and appliance operation to conform to overall energy usage goals.

The new genre of embedded software is characterized by being agile, self-organizing, critically resource constrained, and communication-centric on numerous small devices operating as a collective, rather than highly engineered to a particular stand-alone task on a device sized to suit. The application space is huge, spanning from ubiquitous computing environments where numerous devices on people and things interact in a context-aware manner, to dense in situ monitoring of life science experiments, to condition-based maintenance, to disaster management in a smart civil infrastructure. A common pattern we find is that the mode of operation is concurrency intensive for bursts of activity and otherwise very passive watching for a significant change or event. In the bursts, data and events are streaming in from sensors and the network, out to the network and to various actuators. A mix of real-time actions and longer-scale processing must be performed. In remaining majority of the time, the device must shutdown to a very low power state, yet monitor sensors and network for important changes while perhaps restoring energy reserves. Net accumulation of energy in the passive mode and efficiency in the active mode determine the overall performance capability of the nodes.

To explore the system design techniques underlying these kinds of applications and the emerging technology of microscopic computing, we have developed a series of small RF wireless sensor devices, a tiny operating system (TinyOS), and a networking infrastructure for low-power, highly constrained devices in dynamic, self-organized, interactive environments. The hardware platform grew out of the 'Macromote' developed in SmartDust project as a demonstration of the current analog of what might be put into a cubic millimeter by 2005 [8]. Our first experimental platform (Figure 1) had a 4MHz Atmel AVR 8535 Microcontroller, 8 KB of program store, 0.5 KB of SRAM, a single-channel low power radio [9], EEPROM secondary store, and a range of sensors on an expansion bus [4]. It operates at about 5 mA when active and 5μA in standby, so a pair of AA batteries provides over a year of lifetime at 1% active duty cycle. The severe resource constraints put this platform far beyond reach of conventional operating systems. TinyOS is a simple, component-based operating system, which primarily is a framework for managing concurrency in a storage and energy limited context. A collection of modular components build up from modulating the radio channel and accessing sensors via ADCs to an event-driven environmental

monitoring application with dynamic network discovery and multihop ad hoc routing. A non-blocking discipline is carried through out the design and most components are essentially reentrant cooperating state machines.

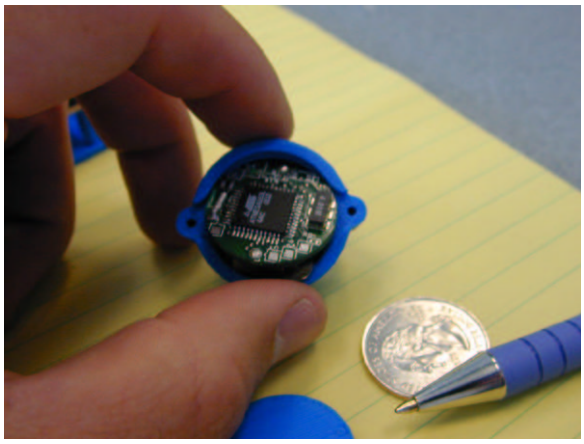

Figure 1. The DOT mote.

The remainder of this paper describes the communication-centric design issues that have arose in developing TinyOS and its tiny networking stacks. Section 2 provides background and a general introduction to TinyOS. Section 3 introduces the Tiny Active Message communication abstraction and illustrates how it is used to form high level networking capabilities. Section 4 examines a collection of software challenges underlying the communication abstraction. Section 5 provides a brief performance evaluation and Section 6 outlines future directions.

2 TinyOS Concepts

Tiny OS, like conventional operating systems, seeks to reduce the burden of application development by providing convenient abstractions of physical devices and highly tuned implementations of common functions. However, this goal is especially challenging because of the highly constrained resource context, the unusual and application specific character of the devices, and the lack of consensus on what layers of abstraction are most appropriate in this regime. The TinyOS approach is to define a very simple component model and to develop a range of components, subsets of which are composed to support a particular application. With time and experience, the new layers of abstraction are likely to emerge. The TinyOS component model focuses on providing a rich expression of concurrency within limited resources, rather than interface discovery or format adaptation [1,10]. Since the deeply embedded sensor networks that we target must run unattended for long periods, robustness is essential. The component

model with narrow interfaces is a significant aid, but given the processing, storage, and energy constraints, we need very efficient modularity. The programming model provides extensive static information, so that compile-time techniques can remove much of the overhead associated with a modular approach and eventually can provide unusual analyses, such as jitter bounds.

A complete TinyOS application consists of a scheduler and a graph of components. Each component is described by its interface and its internal implementation, in a manner similar to many hardware description languages, such as VHDL and Verilog. An interface comprises synchronous *commands* and asynchronous *events*. We think of the component as having an upper interface, which names the commands it implements and the events it signals, and a lower interface, which names the commands it uses and the events it handles. The implementation is written entirely in terms of the interface name space. A component also has internal storage, structured into a *frame*, and internal concurrency, in the form of very light-weight threads, called *tasks*. The command, event, and task handlers are declared explicitly in the source. The points where an external command is called, event is signaled, or task is posted are also explicit in the static code, as are references to frame storage. A separate application description describes how the interfaces are 'wired together' to form the overall application composition. The wiring need not be 1-1; an event may be delivered to multiple components or multiple components may use the same command. Thus, although the application is extremely modular, the compiler has a great deal of static information to use in optimizing across the whole application, including the operating system. In addition, the underlying run-time execution model and storage model can be optimized for specific platforms. A typical application graph is shown in Figure 2, containing a low-power radio stack, a UART serial port stack, sensor stacks, and higher level network discovery and ad hoc routing to support distributed sensor data acquisition. This entire application occupies about three kilobytes.

The TinyOS concurrency model is essentially a two-level scheduling hierarchy - events preempt tasks, tasks do not preempt other tasks. The philosophy is that the vast majority of operation is in the form of non-blocking state transitions. Inter-component operation in tasks is relatively familiar. Within a task, commands may be called, a command may call subordinate commands, or it may post tasks to continue working logically in parallel with its invocation. By convention, all commands return a status indicating whether the command was accepted, providing a full handshake. Since all components have bounded storage, a component must be able to refuse commands. It is very common for a command to merely initiate an operation, say accessing a sensor or sending a message, leaving the operation to be carried out concurrently with other activities, either using hardware parallelism or tasks.

Events are initiated at the lowest level by hardware interrupts. Events may signal higher level events, call commands, or post tasks. Commands cannot signal events. Thus, an individual event may propagate through multiple levels of components, triggering collateral activity. Whenever the work cannot be ac-

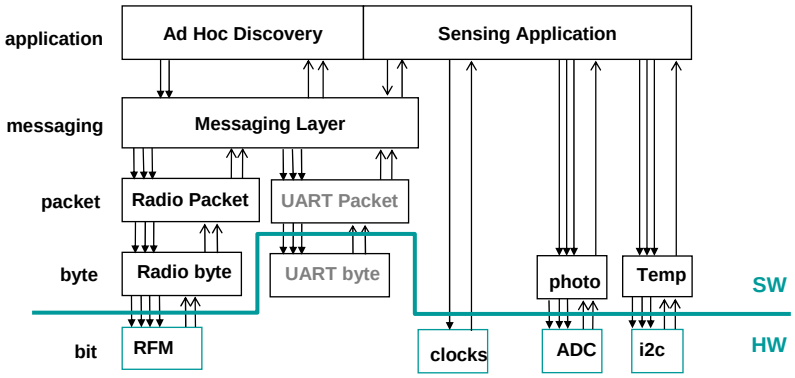

Figure 2. Typical networking application component graph.

complished in a small, bounded amount of time, the component should record continuation information in its frame and post a task to complete the work. By convention, the lowest level hardware abstraction components perform enough interrupt processing to reenable interrupts before signaling the event. Events (or tasks posted within events) typically complete the split-phase operations initiated by commands, signaling the higher-level component that the operation has completed and perhaps passing it the data.

A non-blocking approach is taken throughout TinyOS. There are no locks and components never spin on a synchronization variable. A lock-free queue data structure is used by the scheduler. Components perform a phase of an operation and terminate, allowing the completion event to resume their execution. Most components are written essentially as reentrant state machines. Currently, TinyOS is written in C with conventional preprocessor macros to highlight the key concepts. Better linguistic support would be natural and desirable as the approach becomes established. In our current implementation, the TinyOS execution model is implemented on a single shared stack with a static frame per component.

To make the discussion concrete, Figure 3 shows a TinyOS C code fragment that periodically obtains a value from a sensor and communicates it to neighboring nodes, which render it on their LEDs. The top level structure declares the component storage frame, a command handler, and four event handlers, one of which handles a message event. Each declaration is decorated with a TOS handler type and each would be reflected in the external interface. Here the frame

provides an outgoing message buffer for the component and a state variable. The CHIRP_INIT command illustrates the synchronous module interface; it invokes a subordinate command in the clock module to request periodic events, one per second. A local name is used for the clock initialization command, which is bound to the name used within the particular clock component by the application description graph. Similarly, a local event handler (CHIRP_CLOCK_EVENT) is wired to the clock output event. Here the clock event initiates acquisition of a sensor data value, unless the previous data has not yet been transmitted. The reference to a frame variable is explicit, using VAR.

```
TOS_FRAME_BEGIN(CHIRP_frame) {
  TOS_Msg msg;          /* Message transmission buffer */
  char send_pending; /* State of buffer*/
}
TOS_FRAME_END(CHIRP_frame);

char TOS_COMMAND(CHIRP_INIT)(){
     return TOS_CALL_COMMAND(CHIRP_CLOCK_INIT)(tick1ps);
}

void TOS_EVENT(CHIRP_CLOCK_EVENT)(){
  if (VAR(send_pending) == 0) return TOS_CALL_COMMAND(CHIRP_GET_DATA)();
}

char TOS_EVENT(CHIRP_DATA_EVENT)(int data){
  VAR(msg).data[0] = (char)(data >> 2) & 0xff;
  VAR(send_pending) = 1;
  if (TOS_CALL_COMMAND(CHIRP_SEND_MSG)(TOS_BCAST_ADDR,AM_MSG(CHIRP_MSG), &VAR(data))
    return 1;
  }else {
    VAR(send_pending) = 0;
    return 0;
  }
}

char TOS_EVENT(CHIRP_MSG_SEND_DONE)(TOS_MsgPtr msg){
  if(&VAR(msg) == msg) VAR(send_pending) = 0;
  return 1;
}

TOS_MsgPtr TOS_MSG_EVENT(CHIRP_MSG)(TOS_MsgPtr msg){
  TOS_CALL_COMMAND(CHIRP_OUTPUT(msg->data[0]);
  return msg;
}
```

Figure 3. Example Tiny Active Message application.

The sensor data acquisition protocol illustrates a common, split-phase access pattern. The command causes the operation to start. The component will finish a phase of its work and return, typically allowing the task or event to complete. When the operation is complete, a corresponding event is fired. Here, the ADC runs concurrently and signals the data ready event, which converts the 10-bit sensor value to an 8-bit quantity and requests that it be transmitted in a message. Completion of the transmission operation will signal the CHIRP_MSG_SEND_DONE event on the local node. Arrival of the message at neighboring nodes will signal the CHIRP_MSG event on those nodes, passing it the message data. Details of the messaging portions are described below.

3 Application-Level Communications Challenges

A key test of the communication-centric design approach in TinyOS is its utility in constructing a networking infrastructure for self-organized, deeply embedded collections of devices. We outline several of the key challenges that arise within the networking 'stack' and describe how they are addressed within TinyOS. We begin with the application level messaging model, a variant of Active Messages [11]. Going upward, we describe a simple dynamic network discovery and ad hoc multihop routing layer. Going down, the next section addresses a number of the detailed issues in implementing such a stack in few instructions, little storage, and very little power.

3.1 Tiny Active Messages

Active Messages (AM) is a simple, extensible paradigm for message-based communication widely used in large parallel and distributed computing systems [6,11]. At its core is the concept of overlapping communication and computation through lightweight remote procedure calls. Each message contains the name of a 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. The AM communication model is especially well-suited to the execution framework of TinyOS, as it is event-driven and specifically designed to allow a very lean communication stack to process packets directly off the network, while supporting a wide range of applications.

Initiating an Active Message involves four components, specifying the data arguments, naming the handler, requesting the transmission, and detecting transmission completion. Receiving involves invoking the specified handler on a copy of the transmitted data. This family of issues is illustrated in Figure 3.

The SEND_MSG command identifies intended recipients (here using the broadcast address for the local cell of nodes that pick up the radio transmission), the handler that will process the message on arrival, (here CHIRP_MSG), and the source output message buffer in the local frame. A handler registry is maintained, and TOS_MSG extracts the identifier for the named handler. The

status handshake for this command illustrates the general notion of components managing their bounded resources. The messaging component may refuse the send request, for example, if it is busy transmitting or receiving a message and does not have resources with which to queue the request. The reaction to this occurrence is application specific; in this example we forgo transmitting the particular sensor reading. We might instead adjust data acquisition frequency or phase.

The message arrival event is similar to other events. One key difference is that the Active Message component dispatches the event to the component with the associated message handler. Many components may register one or more message handlers. Additionally, the input to the handler is a reference to a message buffer provided by the Active Message component.

3.2 Managing Packet Buffers

Managing buffer storage is a sticky problem in any communication stack. Traditional operating systems push this complexity into the kernel, providing a simple, unlimited user model at the cost of copying, storage management complexity, and blocking interfaces. High performance models, such as MPI, define a suite of send and receive operations with distinct completion semantics[3]. Three issues must be addressed: encapsulating useful data with transport header and trailer information, determining when output message data storage can be reused, and providing an input buffer for an incoming message before the message has been inspected to determine where it goes. The Tiny Active Message layer provides simple primitives for resolving these issues with no copying and very simple storage management.

The message buffer has a defined type in the frame that provides holes for system specific encapsulation, such as routing information and error detection. These holes are filled in as the packet moves down the stack, rather than following pointers or copying. The application components refer only to the data field or the entire buffer. References to message buffers are the only pointers carried across component boundaries in TinyOS.

Once the send command is called, the transmit buffer is considered 'owned' by the network until the messaging component signals that transmission is complete. The mechanism for tracking ownership is application specific; our example maintains a pending flag. Since a strict ownership exchange is involved, no mutex is required around updates to the flag, although some care in coding must be exercised. If the send command is accepted, the SEND_DONE event will asynchronously clear the pending flag.

Observe that the SEND_DONE event receives a reference to the completed buffer as an argument and must check whether the buffer is its own. This event is delivered to all components that register AM handlers. A component that receives a done signal for another's transmission may use the event to retry a previously refused request.

The message handler receives a reference to a 'system owned' buffer, which is distinct from its frame. The typical behavior is to process information in the

message and return the buffer, as in our example. In general, the handler must return a reference to *some* free buffer. It could retain the buffer it was given by the system and return a different buffer that it 'owns'. A common special case of this scenario is a handler that makes a small change to an incoming message and retransmits it. We would like to avoid copying the remainder of the message. However, we cannot retain ownership of the buffer for transmission and return the same buffer to the system. Such a component should declare a message buffer and a message buffer pointer in its frame. The handler modifies the incoming buffer and exchanges buffer ownership with the system. If its previous transmit buffer is still busy, one of the two operations must be discarded. A component performing reassembly from multiple packets may own multiple such buffers. In any case, runtime buffer storage management is reduced to a simple pointer swap.

3.3 Network discovery and ad hoc routing

A more sophisticated use of the tiny Active Message model is illustrated by its use in supporting dynamic network discovery and multihop ad hoc routing. Discovery could be initiated from any node, but often it is rooted at gateway nodes that provide connectivity to conventional networks. Each root periodically transmits a message carrying its ID and its distance (zero) to its neighborhood. The message handler checks whether the source is the 'closest' node it has heard from recently (i.e., in the current discovery phase) and, if so, records the source ID as its multihop parent, increments the distance, and retransmits the message with its own ID as the source. The discovery component utilizes the buffer swap described above. Observe, this simple algorithm builds a breadth first spanning tree in a distributed fashion rooted at the original source. Each node records only a fixed amount of information. The specific shape of the tree is determined by the physical propagation characteristics of the network, not any prespecified layout, so the network is self-organizing. With multiple concurrent roots, a spanning forest is formed.

Routing packets up the tree is straightforward. A node transmitting data to be routed specifies a multihop forwarding handler and identifies its parent as the recipient. The handler will fire in each of its neighbors. The parent retransmits the packet to its parent, using the buffer swap. Other neighbors simply discard the packet. The data is thus routed hop-by-hop to the root. Reduction operators can be formed by accumulating data from multiple 'children' before transmitting a packet up the tree.

The discovery algorithm is non-optimal because of redundancy in the outgoing discovery wave front and might be improved by electing cluster leaders or retransmitting the beacon with some probability inversely related to the number of siblings. Alternatively, the discovery phase can be eliminated entirely by piggybacking the distance information on the sensor data messages. When a node hears a packet from a node fewer hops from the base station, it adopts the source as its parent. The root node simply transmits a packet to itself to grow

the routing tree. (Nodes must also age their current distance to adapt to changes in network topology due to movement or signal propagation changes.)

These simple examples illustrate the fundamental communication step upon which distributed algorithms for embedded wireless networks are based: receiving a packet, transforming it, and selectively retransmitting it or not. Squelching retransmission forms an outgoing wave front in discovery and forms a beam on multihop routing. In these algorithms the data structure for determining whether to retransmit is little more than a hop count, more generally it might be a cache of recent packets [5,7].

4 Lower-level Communication Challenges

This section works down from the messaging component to illustrate the nuts and bolts of realizing the lower layers if a tiny communications stack.

4.1 Crossing layers without buffering

One challenge is to move the message data from the application storage buffer to the physical modulation of the channel without making entire copies, and similarly in the reverse direction. A common pattern that has emerged is a cross layer 'data pump'. We find this at each layer of the stack in Figure 2. The upper component has a unit of data partitioned into subunits. It issues a command to request transmission of the first subunit. The lower component acknowledges that it has accepted the subunit and when it is ready for the next one it signals a subunit event. The upper handler provides the next unit, or indicates that no more are forthcoming. Typically this is done by calling the next subunit command within the ready handler. The message layer is effectively a packet pump. The packet layer encodes and frames the packet, pumping it byte-by-byte into the byte layer. On the UART, the byte-by-byte abstraction is implemented directly in hardware, whereas on the radio the byte layer pumps the data bit-by-bit into the radio. Each of these components utilizes the frame, command, and event framework to construct a re-entrant software state machine.

4.2 Listening at low power

In traditional remote control or remote monitoring applications, a well-powered stationary device is always receiving and a portable device transmits infrequently. However, in a multi hop data collection network, each node will transmit its own data from time to time and listen the rest of the time for data that it needs to forward toward a sink.

Although active transmission is the most power intensive mode, most radios consume a substantial fraction of the transmit energy when the radio is on and receiving nothing. In ad hoc networks, a device will only be transmitting for short periods of time but must be continually listening in order to forward data for the surrounding nodes. The total energy consumption of a device ends up being

dominated by RF reception cost. To address this, we employ two techniques to reduce the power consumption while listening.

A fairly traditional way to accomplish this is through a technique we refer to as *periodic listening*. By creating time periods when it is illegal to transmit, nodes must listen only part time. This approach works well when the time scale of the invalid periods is quite large relative to the message transmission time. For example, we have an implementation of this mechanism where the transmission window is ten seconds and the sleep window is 90 seconds. This reduces the reception power consumption of the nodes by approximately 90%. However, downside of this simple approach is that it limits the realized bandwidth available by the same factor.

The reduction of network bandwidth into a mobile node is often unacceptable in the context of sensor networks. Any node may be act as a router or data processing point and need to fully utilize the radio bandwidth. To address this issue we have developed second technique that we call *low power listening*. This method keeps the same listener duty cycle concept, but greatly reduces the time scale. Each receiver turns its radio on for 30 μs out of a 300 μs window instead of 10 sec out of 100 sec. This permits the same 90% energy savings as periodic listening yet does not decrease the available channel capacity. The downside of this method is that a transmitter must spend extra energy to make sure that the receiver has its radio on before packet transmission begins. A transmitter must send a packet preamble designed to get the attention of the receiver. Because the sender knows that the receiver will be listening every 300μs, the preamble must be at least the same duration. In our system, the data packet length is 56,100 μs long, so preamble overhead is quite small compared to the transmission cost. A 90% decrease in idle power consumption is gained with less than a 1% increase in transmission cost without changing the channel capacity. In the case of a node that transmits one packet per second and receives one packet per second from other nodes, there is a net power reduction for the radio of approximately 75%. System level power measurements on real hardware have confirmed this power savings.

Operation	Time	Normal	Low Power Mode
Transmit	50 ms/50.5 ms	600uj	606uj
Receive	50 ms	250 uj	250 uj
Listen	900 ms	4500 uj	450 uj
Total	1000 ms	5350 uj	1306 uj
Savings			75%

Table 1. Power savings breakdown for a radio that receives and transmits 1 packet per second with a TX power consumption of 12mA and an RX power consumption of 5mA. Packet transmission and reception takes 50ms.

To further reduce the average power consumption of the network, low power listening can be combined with the periodic listening. Running both schemes si-

multaneously results in listening at reduced power for only a fraction of the time. The power reductions are multiplicative. These techniques provide a mechanism for trading bandwidth and transmission cost for a reduction in receive power consumption.

4.3 Physical layer interface

Traditional I/O subsystems have a controller hierarchy that abstracts the device specific characteristics and timing requirements of the physical layer from the main system controller. In contrast, our hardware directly connects the central microcontroller to the radio. This places all of the real time requirements of the radio onto the microcontroller. It must handle every bit that is transmitted or received in real time. Additionally, it controls the timing of each bit so that any jitter in the control signals that it generates is propagated to the transmitted signal. The TinyOS communication stack has been constructed to handle these constraints while allowing higher level functions to continue in parallel.

At the base of our component stack is a state machine that performs the bit timing. The RFM component transfers a single bit at a time to and from the RF Monolithics radio. For a correct transmission to occur, the transmitted bit must be placed and held on the TX line of the radio for exactly one bit time. In our system that is 100μs. For reception, the RX line of the radio must be sampled at the midpoint of the transmission period. The radio provides no support for determining when bit times have completed.

The interface to our RFM component takes the form of a data pump. It orchestrates a bit-by-bit transfer from a byte level component to the physical hardware. To start the transmission of data, a command is issued to the RFM component to switch into transmit mode. Then a second command is used to transfer a single bit down to the RFM component. This bit is immediately placed onto the transmit line. After 100μs has passed, the RFM component will signal an event to indicate that it is ready for another bit. The byte level components response is to issue another command to the RFM component that contains the next bit. This interaction of signaling and event and receiving the next bit continues until the entire packet is completed. The RFM layer component abstracts the real time deadlines of the transmission process from the higher layer components.

During transmission, complex encoding must be done on each byte while simultaneously meeting the strict real time requirements of the bit layer. The encoding operation for a single byte takes longer than the transmission time of single bit. To ensure that the encoded data is ready in time to meet the bit level transmission deadline, we must start the encoding of the next byte prior to the completion of the transmission of the current byte. We use the TinyOS task mechanism to execute the encoding operation while simultaneously performing the transmission of previous data. By encoding data one byte in advance of transmission, we are using buffering to decouple the bit level timing from the byte encoding process.

Data reception takes the same form as transmission except that the receiver must first detect that a transmission is about to begin and then determine the timing of the transmission. To accomplish this, when there is activity on the radio channel, the RFM layer component is set to sample bits every 50μs, double sampling each byte. These bits are handed up one at a time to the byte level component. The byte level component creates a sliding buffer of these bit values that contains the last 18 bits. When the value of the last 18 bits received matches the designated start symbol, the start of a packet has been detected. Additionally, the timing of the packet has been determined to within half a bit time. Next, the RFM layer is told to sample a single bit after 75μs. This causes the next sample to fall in the middle of the next bit window, half way between where the double sampling would have occurred if the sample period had remained at 50μs. Finally, the RFM is told to sample every 100μs for the remainder of the packet.

4.4 Media Access and Transmission Rate Control

In wireless embedded systems, the communication path to the devices is not a dedicated link as it is in most traditional embedded system, but instead a shared channel. This channel represents a precious resource that must be shared effectively in the context of resource constrained processing and ad hoc multihop routing. Moreover, many applications require that nodes have roughly equal ability to move data through the network, regardless of position within the network topology. We have extended the low-level TinyOS communication components with an energy-aware media access control (MAC) protocol and developed a simple technique for application specific adaptive rate control [12].

Since the I/O controller hierarchy on our small devices is so primitive, the MAC protocols must be performed on micro-controller concurrently with other operations. The RF transceiver lacks support for collision detection, so we focus on Carrier Sense Multiple Access (CSMA) schemes, where a node listens for the channel and only transmits a packet if the channel is idle. The mechanism for clocking in bits at the physical layer is also used for carrier sensing. Thus, the MAC layer is implemented at both the bit and byte level in the network stack. If consecutive sampling of the channel discovers no signal, the channel is deemed idle and a packet transmission is attempted. However, if the channel is busy, a random back off occurs. The entire process repeats until the channel is idle. A simple 16-bit linear feedback shift register is used as a pseudo random number generator for the back off period. Since energy is a precious resource, the radio is turned off during back off. Many applications collect and transmit data periodically, perhaps after detecting a triggering event, so traffic can be highly correlated. Detection of a busy channel suggests that a neighboring node may indicate that the communication patterns of the nodes are synchronized. The application uses the failure to send as feedback and shifts it sampling phase to potentially desynchronize. This simple scheme has been shown to yield 75% channel utilization for densly populated cell.

Another common application requirement is roughly equal coverage of data sampling over the entire network. In other words, each node in the network should be able to deliver fair allocation of bandwidth to the base station. With our ad hoc routing layer, nodes self organize into a spanning forest, where each node originates and routes traffic to a base station. The competition between originated and route-thru traffic for upstream bandwidth must be balanced in order to meet the fairness goal. Furthermore, given that the capacity of a multi-hop network is limited [2], nodes must adapt their offered load to the available bandwidth rather than over-commit the channel and waste energy in transmitting packets that can never reach the base station. Our adaptive transmission control scheme is a local algorithm implemented above the Active Message layer and below the application level. The application has a baseline sampling rate that determines in maximum transmission rate and transmits a sample with a dynamically determined probability. On successful transmission the probability is increased linearly, whereas on failure it is decreased multiplicatively. A successful transmission can be indicated by an explicit acknowledgment from the receiver or an implicit acknowledgment when the sender hears its packet being forwarded by its parent. Since implicit acknowledgment is often application specific, the application decides if the transmission was successful and propagates the information down to the transmission control layer. Rejection of application's transmission command at the transmission control level triggers the adaptation.

5 Evaluation

To demonstrate the performance of the Active Messages model, we performed single message round-trip timings (RTT) and measured energy overhead on a sample implementation. The measurements were made on embedded sensors with a 4MHz Atmel AVR 8535 Microocontroller and an RFM radio. The Tiny Active Messages software component consumes 322 bytes of a 2.6KB total binary image. At a 10kbps raw bit rate and using a 4b6 encoding scheme, the wireless link supports 833 bytes/sec of throughput. Figure 4 presents the RTT results for various lengths through a network. A route length of one measures a host computer to base station time (40ms) and reflects the cost of the wired link, device processing, and the host OS overhead. For routes greater than one hop, the RTT 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. Table 2 presents a cumulative profile of the single hop RTT. 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.

Exploiting the event based nature of Active Messages enables a high degree of power savings. When no communication or computation is being performed, the device enters a low-power idle state. We measured the power consumption for

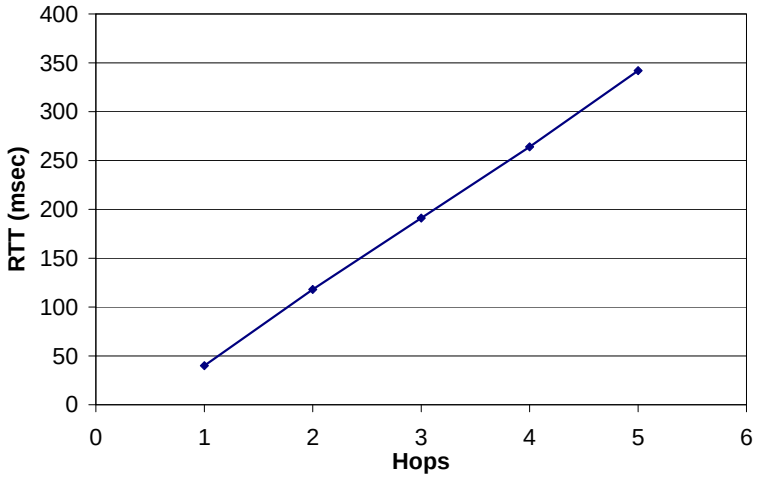

Figure 4. Round trip times for various route lengths. Note that one hop measures the time for a message between the Host PC and base station device.

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 2. Cumulative time profile for a single hop RTT test.

this idle state, the peak power consumption and the energy required to transmit one bit. The results are presented in Table 3.

6 Conclusion

The TinyOS approach has proven quite effective in supporting general purpose communication among potentially many devices that are highly constrained in terms of processing, storage, bandwidth, and energy with primitive hardware support for I/O. Its event driven model facilitates interleaving the processor between multiple flows of data and between multiple layers in the stack for each flow while still meeting the severe real-time requirements of servicing the radio. Since storage is very limited, it is common to process messages incrementally at several levels, rather than buffering entire messages and processing them level-by-level. However, events alone are not sufficient; it is essential that an event be able to hand any substantial processing off to a task that will run outside the real-time window. This provides logical concurrency within the stack and is

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

Table 3. Power and energy consumption measurements.

used at every level except the lowest hardware abstraction layer. By adopting a non-blocking, event-driven approach, we have been able to avoid supporting traditional threads, with the associated multiple stacks and complex synchronization support.

The component approach has yielded not only robust operation despite limited debugging capabilities, it has greatly facilitated experimentation. For example, we had little understanding of what would be the practical error characteristics of the radio channel when the development started, and we ended up building several packet layers that implemented different coding and error detection strategies. The packet components could be swapped with a simple change to the description graph and temporary components could be interposed between existing components, without changing any of the internal implementations. Moreover, the use of components and the TinyOS programming style allows essentially an entire subtree of components to be replaced by hardware and vice versa.

The Tiny Active Message programming model has made it easy to experiment with numerous higher level networking layers and fine-grained distributed algorithms. Although the devices are quite limited, we spend little effort worrying about the low-level machinery while building high-level, often application specific protocols. Several higher level capabilities have recently been developed on this substrate. One example is the ability to reprogram the nodes over the network. A node can obtain code capsules from its neighbors or over multihop routes and assemble a complete execution image in its EEPROM tiny secondary store. The node can then use this to reprogram itself. Other examples include a general purpose data logging and acquisition capability, a facility to query nodes by schema, and to aggregate data from a large number of nodes within the network. We are currently developing mechanisms for operating the radio at five to ten times the bit rate, while keeping all of the higher level structures.

Without the traditional layers of abstraction dictating what kinds of capabilities are available, it is possible to foresee many novel relationships between the application and the underlying system. Our adaptive transmission control scheme is a simple example; rejection of the send request causes the application to adjust its rate of originating data. The application level forwarding of multihop traffic allows the node to keep track of its changing set of neighbors. Moreover, the radio is itself another sensor, since receive signal strength is provided to the ADC. Thus, each packet can be accompanied by signal strength data for use in estimating physical distance or presence of obstructions. The radio is also an actuator, as its signal strength, and therefore cell size, can be controlled.

The lowest layer components are synchronizing all receivers to the transmitter to within a fraction of a bit. Thus, very fine grain time synchronization information could be provided with every packet for control applications. What started as a tremendously constraining environment where traditional abstractions were intractable has become a rich and open playground for experimenting with novel software structures for deeply embedded, networked systems.

References

1. Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.
2. Jinyan Li et. al. Capacity of ad hoc wireless networks. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, Rome, Italy, July 2001.
3. MPI Forum. Mpi: A message passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3-4):169–416, 1994.
4. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.
5. Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah 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*, August 2000.
6. 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.
7. Charles E. Perkins, editor. *Ad Hoc Networking*. Addison-Wesley, New York, NY, 2001.
8. K. S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes. *1999 Electronics Research Laboratory Research Summary*, 1999.
9. RF Monolithics, Inc. Tr1000 916.50 mhz hybrid transciever. `http://www.rfm.com/products/data/tr1000.pdf`.
10. Sun Microsystems, Inc. Jini network technology. `http://www.sun.com/jini`.
11. 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.
12. Alec Woo and David Culler. A transmission control scheme for media acces in sensor networks. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, Rome, Italy, July 2001.