TinyOS – An Operating System for Tiny Embedded Networked Sensors

By

Sharan Raman

Paper Presentation for Advanced Operating Systems Course, Spring 2002

1 . Abstract

This paper discusses the background and application requirements that motivated the development of TinyOS. It enumerates the characteristics associated with any typical Networked Sensor application. The hardware platform that was used for deploying TinyOS is also described. The design aspects of the Event based TinyOS is discussed in detail. This paper also enlightens the Tiny Active Messaging Model used by the TinyOS for data communication and its implementation concepts. The paper goes ahead and describes some application level communication concepts like managing packet buffers, Network discovery and Ad Hoc Routing and Media Access and Transmission Control. Then there is a bunch of discussions that includes topics like Evaluation of TinyOS, Comparison of TinyOS with other Operating Systems and Future Research Work.

2. Introduction

2.1 Background

TinyOS is an *event based* operating environment that is designed for use with *embedded networked sensors* [1] .It is designed to support the *concurrency intensive* operations required by networked sensors with minimal hardware requirements. It uses the *Active Message Communication model* [2] for building *non-blocking* applications and higher Networking capabilities like *Multihop ad hoc routing*. TinyOS was developed by a group of four Computer Science Graduate Students at the *University of California, Berkeley*. The development of TinyOS was supported by Defense Advanced Research Project Agency *(DARPA),* the National Science Foundation *(NSF)* and *Intel Corporation*. [5] [1]

The emergence of compact, low-power wireless communication and Networked sensors is giving rise to entirely new kinds of embedded systems that are *distributed* and deployed in *dynamic*, constantly changing and *adaptive* control environments. These Networked Sensors are compact devices that can be used to sense light, heat, position, movement, chemical presence etc from real environments and communicate information back to traditional computers. They also need to assist each other in collection of data and conveying them back to the centralized collection point. [2]

These Embedded Sensors are characterized to be *agile, self-organizing, critically resource constrained and communication centric*. Their application space is huge including all monitoring applications in a context aware situation, situation monitoring of life science experiments, disaster management and others. There are two design issues associated with this scenario: these devices are *Concurrency Intensive* where several flows of data must be handled simultaneously and the system must provide *Efficient Modularity*, which means that hardware specific and application specific components must coalesce together with little processing and storage overhead [2]. There are *bursts of activity* where data and events stream in from the sensors and the Network and periods of *passive listening* to significant events. During the bursts a mix of real time actions and long-scale processing/computation must be performed and in the remaining majority of time, the device shuts to a very *low power state* and monitors for changes in the system state.

2.2 Networked Sensor Characteristics [1]

- **Small Physical Size and low power consumption**

  Minimal Size and power constrain the processing time, storage and interconnect capacity of the device. Due to these constrained resources, the operating system and applications have to use them efficiently.

- **Concurrency Intensive Operation**

  These sensors have to communicate information with little processing on the fly. Information may be simultaneously captured from sensors, manipulated and streamed onto a network. Data may also be received from other nodes and have to be forwarded to the next hop in the network. Hence the system must handle multiple flows of data concurrently and also perform processing and communication parallely.

- **Limited Physical Parallelism and Controller Hierarchy**

  The Number of *independent device controllers*, their capabilities and complexity of the interconnect are much lower for these Networked Sensors when compared to conventional systems. The sensors provide a primitive interface directly to the central controller unlike conventional systems that distribute concurrent processing over multiple levels of controllers. This *limited hierarchy* is a repercussion of the Resource constraints.

- **Diversity in Design and Usage**

  These devices are *application specific*, rather than general purpose. The hardware is specific to the application and the variations in them are likely to be large. Hence these devices require an unusual degree of *Software Modularity* that must be efficient. A generic development environment is needed which allows the development of specialized applications and allows easy *migration* of components across the software/hardware Boundary.

- **Robust Operation**

  These devices will be *numerous*, largely *unattended* and expected to be operational a large fractional of time. The application of redundancy techniques for fault-tolerance is constrained by space and power limitations. Thus, enhancing the reliability of individual devices is essential.

  As the previous embedded operating systems are more general purpose, they occupy too *much memory* and work on *heavy weight processes*. They also have a *deep hierarchy* of controllers and kernel layers and the *context switch time* to perform different functions is too much. As these Embedded OS do not cater to the needs of the Networked Sensor System, the development of Tiny OS was important.

3. TinyOS Concepts

3.1 Hardware Organization

The UC-Berkeley group developed a small, flexible networked sensor platform that expressed the key characteristics of the general class (in Section 2.2). Figure 1 shows the hardware configuration of the device.

There is a *microcontroller MCU* (ATMEL 90LS8535) that has an *8-bit Harvard Architecture* processor with *16-bit* addresses. It has *32 8-bit* general registers and runs at *4 MHz* and *3.0 Volts*. It has *8 KB* of *Flash Program Memory* and *512 Bytes* of *SRAM* as the data memory. A co-processor is used to write instructions to the Program Memory. It also has a *single-channel low power radio*, an *EEPROM secondary* store and a range of sensors like Light (photo sensor) and Temperature sensors connected to the Bus. [3]

There are 3 power modes that the processor operates on: *idle*, which just shuts off the processor, *power down*, which shuts off everything but the watchdog and asynchronous interrupt logic necessary for wake-up and *power save* which is similar to power down, but leaves a timer also running. The sensors use *Analog to Digital Converters* to communicate data to the processor. [3]

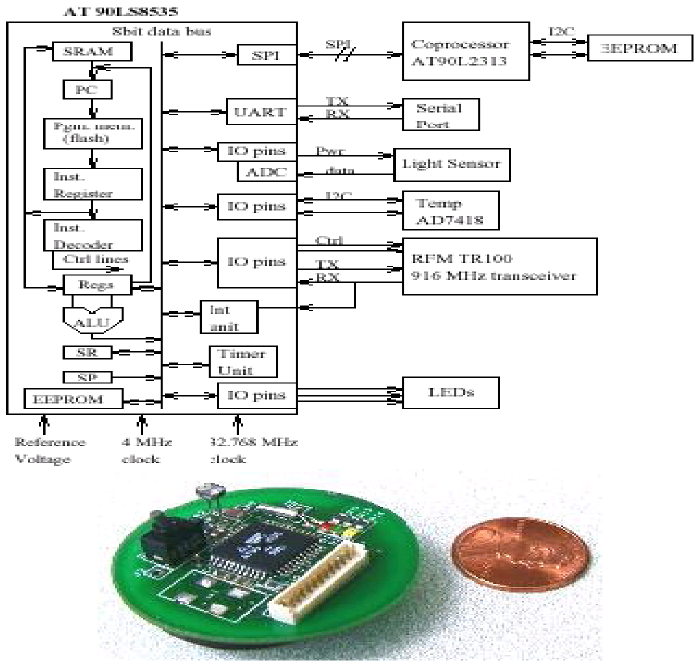


Figure 1: Photograph and schematic for representative network sensor platform [1]

The Radio device contains no buffering and hence each bit must be serviced by the controller on real time. The serial port represents an important asynchronous bit-level device with byte controller support. The main processor can use the coprocessor for extra storage.

The power characteristics of these components in the Networked Sensor show that biggest power savings is achieved by *making unused components inactive* whenever possible. The philosophy of the system should be to get work done as quickly as possible and go to sleep.

There are two types of sensors. One type is a *mobile sensor* that picks up temperature and light readings periodically and presents them to the wireless network. This needs to conserve its limited energy. The other type is a *stationary sensor* that bridges the radio network through the serial link to a host computer. It has a little more power input but has more *demanding data flows*. [3]

3.2 TinyOS Design

TinyOS uses an *Event model* so that high levels of concurrency can be handled in a very small amount of space unlike the *stack based threaded* approach that uses too much stack space and also has a high context switch time. [1]

Since Power is a precious resource, CPU resources must be utilized efficiently. The event-based approach handles tasks associated with events rapidly without *allowing blocking or polling*. Unused CPU cycles are spent in sleep state as opposed to actively looking for events. TinyOS was developed in *C*. [3]

❑ Components, Commands, Events and Tasks

TinyOS is divided into a collection of *Software Components*. A TinyOS application consists of a *scheduler* and *a graph of components* describing their interaction.

A Component has four parts: a set of *Command Handlers*, a set of *Event Handlers*, an encapsulated *fixed-size frame* and a bundle of *simple tasks*. Each component declares the commands it uses and events it signals. [1] [7]

The fixed sized frames are *statically* allocated which helps to know the memory requirements of a component at compile time. The frame is an *internal storage space* that contains the state of the component and is used by the events, commands and tasks. [1] [7]

Each Component is described by its *interface* and its *internal implementation*. An interface contains commands and events. These declarations are used to compose the modular components and this composition creates layers of components that are application specific. The higher-level components issue *commands* to lower-level components while the lower ones *signal events* to the higher-level components. Hence we can think of the component to have an upper interface, which names the *commands it implements* and the *events it signals* a lower interface which names the *commands it uses* and *events it handles*. [3] [7]

Commands are *non-blocking requests* made to lower level components. A command will deposit *request parameters* into its frame and conditionally *post a task* for a later execution. It also provides feedback to its caller (from a higher level component) by *returning status* of success or failure. [3]

Event Handlers are invoked to deal with Hardware events either directly or indirectly. The lowest level components have handlers connected directly to hardware interrupts. An event Handler can deposit information in its frame, post tasks, signal higher-level events or call lower level commands. Events help in forwarding changes upwards while commands forward processing downwards. In order to avoid cycles, *command cannot signal events*. [1]

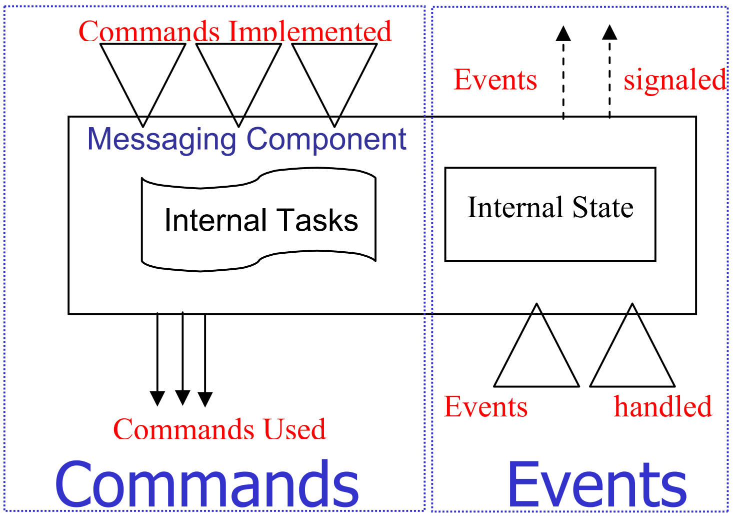


Figure 2: Structure of a Component in TinyOS [6] [7]

Tasks perform the work and are atomic with respect to other tasks. They run to completion and can call lower commands, signal higher-level events and schedule other tasks within the same component. The *run-to completion* property helps to allocate a single stack to the currently executing task and this conserves space. Tasks also allow concurrency within each component as they execute asynchronously. They must never block to avoid delaying progress in other components. Hence we can look at these tasks as blocks of computation. [3]

The *Task scheduler* is a simple *FIFO* scheduler that has a bounded size scheduling data Structure. It is *power sensitive* and puts the processor to sleep when the task queue is empty, but leaves the peripherals operating to wake up the system in case of any new hardware event. There is a *two level scheduling hierarchy* in the TinyOS – *events preempt tasks but tasks do not preempt other tasks*. Since all components have bounded storage, a component has to refuse commands. [1]

4. Application Level Communication Concepts

4.1 Tiny Active Messages

Active Messages (AM) is a simple, extensible paradigm for *message-based* communication widely used in large parallel and distributed computing systems. Each Active Message contains the name of a *user-level* handler to be invoked upon a *target node* and a data payload to pass arguments. The handler serves the dual purpose of extracting the message from the network and performing computation on the data or sending a response message. It overlaps communication and computation through lightweight *remote procedure calls*. [3]

This AM communication model is well suited to the execution framework of the TinyOS, as it is *event-driven* and designed to allow a *very lean communication stack* to process packets directly off the network. Message Handlers must be able to *execute quickly* and *asynchronously.* Initiating an Active Message involves four components: *specifying the data arguments, naming the handler, requesting the transmission and detecting transmission completion.* [2]

❑ Implementation of Tiny Active Messages

TinyOS commands are used to initiate message transfers and this fires off events to message handlers. There is also an event associated with the completion of transmission.

Send Command includes the destination address, handler ID and message body. The Active Message Component in the TinyOS performs *address checking* and *dispatch* and relies on lower components for basic packet transmission. Reliable, error free delivery is *not* guaranteed. There are different levels of error detection and correction in packet level components. These include *basic transmission* without any error detection or correction, *CRC checked packets* that have error detection, and *forward error packets* that provide basic error correction and detection. [4]

A *Host PC package* consists of libraries that can be used to communicate over the PC's serial port to a special base station sensor that has both a *RS232C communication* channel as well as *RF communication*. These libraries help to bring the collected data from the Sensor Networks to a more traditional computing environment. [3]

❑ Packet Format

A *fixed size 30-Byte* packet is used for packet transmission. The First two bytes of a received packet are used to identify the *destination* of the packet (R0) and the ID of the message handler that is to be invoked on the packet (H0). The TinyOS 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*. [4]

❑ Multi-hop Packet Format

The research group designed a Multi-Hop Packet format for source based multi-hop routing applications. A Multi hop packet format is shown below:

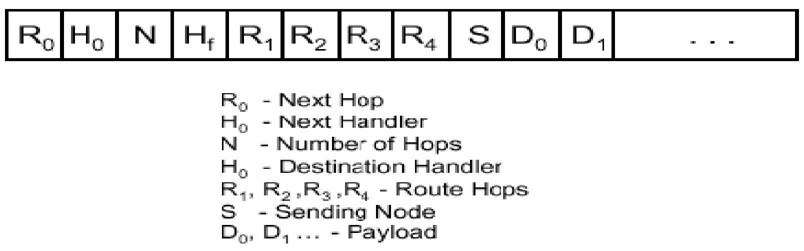

R_0 H_0 N H_f R_1 R_2 R_3 R_4 S D_0 D_1 . . .
R_0 - Next Hop
H_0 - Next Handler
N - Number of Hops
H_0 - Destination Handler
R_1, R_2, R_3, R_4 - Route Hops
S - Sending Node
D_0, D_1 ... - Payload

Figure 3: Multi-hop packet format [4]

This format, 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 (R1, R2, R3, R4): 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 (HF). In this instance, the multi-hop router is simply the handler of a typed message. [4]

While the packet is in-route, H0 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, HF, into H0. [3]

Two special addresses are defined. The first special address needed was the *broadcast address (0xff)*. The concept of a one-to-all broadcast greatly simplifies the route discovery and exploration algorithms. 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. [4]

In TinyOS, multiple applications will use a *single messaging layer*. The Active Message layer can handle only *a single message* at a time. If a message transmission is in progress, a request to transmit an additional message will be *denied*. An application must then retry transmission at a later time. In many data collection applications, it may be better to simply *throw* away the message and wait to transmit the next sensor readings. The Active Messages layer *cannot receive* a message *while a transmission is in progress*. Once a message transmission begins, it runs to completion. [3]

When an application sends out a message, it provides the Active Messages layer with a *pointer* to a memory buffer. The data in this buffer must remain *unchanged* until the transmission is complete. This means that the application cannot modify the buffer until the Active Messages layer has fired the send done event. [3]

When an application receives an incoming message, the messaging layer provides a pointer to the buffer where the message is stored. This data is only guaranteed to be valid for the duration the event that delivered the message. If the application needs to keep the data longer, it must *copy* the data elsewhere. Once the event handler returns, the messaging layer will receive subsequent messages into the same buffer. This will *overwrite* the original message. [3]

4.2 Managing Packet Buffers

Managing buffer storage is very important in any communication stack. 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. [2]

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. 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*. Since a strict ownership exchange is involved, *no mutex is required.* The TinyOS attempts to conserve buffer memory by *reusing* buffer space for multiple applications. [2]

4.3 Network Discovery and Ad Hoc Routing

The TinyOS Active message Model is particularly useful in *Network Discovery* and *Ad hoc routing* applications.

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 and, if so, records the source ID as its multi hop parent, increments the distance, and retransmits the message with its own ID as the source. Each node records only a fixed amount of information. [2]

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 to 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. [2]

5. Lower Level Communication Challenges

5.1 Crossing layers without buffering

Because of the memory constraints, the message data from the application storage buffer has to be moved to the physical modulation of the channel without making entire copies, and similarly in the reverse direction. [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. Thus the same buffer is *re-used* from one layer to another without copying. The same holds for building up data from the lower level components to the higher level ones. [3]

5.2 Listening at Lower Power

In remote monitoring applications, a well-powered stationary device always receives while a mobile sensor device transmits infrequently. In Multi hop data collection network, each node transmits data periodically and listens to the network the rest of the time. [2]

Active transmission is the *most power intensive mode*. There are two techniques that reduce power consumption while listening: Periodic Listening and Low Power Listening. [2]

Periodic Listening creates time periods when it is *illegal* to transmit and the hence nodes need to listen only during some periods. For ex. we can have a transmission window of 10 sec and a sleep window of 90 sec. Hence power consumption can be reduced by 90%.This scheme works well if the invalid period duration is much longer than the message transmission time. But this scheme *limits the realized bandwidth* as there are always some specific periods when nothing is transmitted though transmission data might be available then. [2]

In the *low power listening*, the same concept of selective transmission is used, but the cycle of *window* is very small. For ex. we can have a transmission window of 30 µsec and a sleep window of 270µsec. This has the same power savings and also does not limit the bandwidth too much as the window is too small to miss out any buffered data. But a transmitter now has to *expend extra effor*t to make sure it has a transmission window currently before it can transmit. A *hybrid* of Low power listening and periodic listening can also be used to conserve power.

5.3 Physical Layer Interface

In the TinyOS system the hardware layer directly connects to the Central Microcontroller. Hence the microcontroller has to handle every bit that is transmitted or received in *real time.*

A *state machine component* [2] is designed to perform the bit level timing. From the lower components bit level data is received using this state machine timing and assembled into a byte at the higher component. This Byte is now transmitted bit by bit to its higher component.

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*, the encoding of the next byte must start prior to the completion of the transmission of the current byte*. The TinyOS task mechanism is used to execute the encoding operation while *simultaneously* performing the transmission of previous data. This is similar to byte pre-fetching in case of Instruction set architectures. [2]

5.4 Media Access and Transmission Control

The communication path in wireless embedded systems is not a dedicated link. Hence this precious resource must be *shared effectively* and all nodes in a network should have a fair share of the media, irrespective of their location in the network topology. The TinyOS has an *energy-aware media access control protocol* and also an application specific adaptive rate control for ensuring fair share of the media. [2]

The MAC protocols are performed on the Microcontroller concurrently with other operations, as there is no stack hierarchy. *Carrier Sense Multiple Access (CSMA) protocol* is used where a node listens for the channel and transmits only if the channel is *idle*. The bit-clocking mechanism at the physical layer is also used for carrier sensing. [2]

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 shift register* is used as a *pseudo random number generator* for the back off period. The processor moves into the *low-power mode* during this back off period.

Equal coverage of data sampling over the entire network is very important. Each Node in the network should be able to deliver fair allocation of bandwidth to the base station.

The *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 its *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. This way the transmission is done in an adaptive manner considering the network congestion. [2]

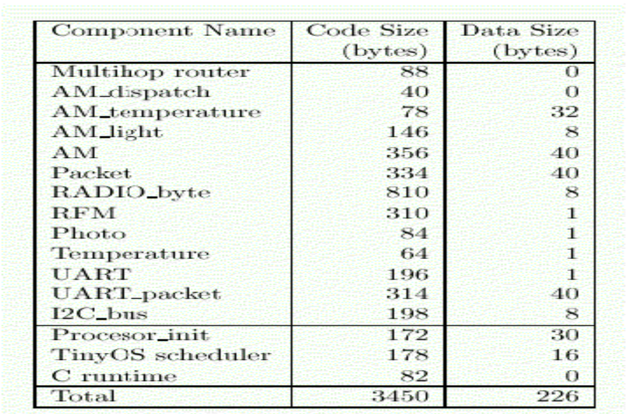
6. Discussions

6.1 Evaluation of TinyOS

- **Small Physical Size**

The source code size for various components of the TinyOS system and the sample Multi hop routing application is shown below. The important TinyOS component '*scheduler'* occupies only *178 Bytes*. The data size of the scheduler is only 16 bytes and it utilizes only 3% of the available data memory. [1]

Software Footprint refers to the total number of bytes occupied by a software component on the device. 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. 40 Bytes is used for static data. Hence the software footprint of the TinyOS is *very small* and this is very useful when memory is strictly constrained. [1]



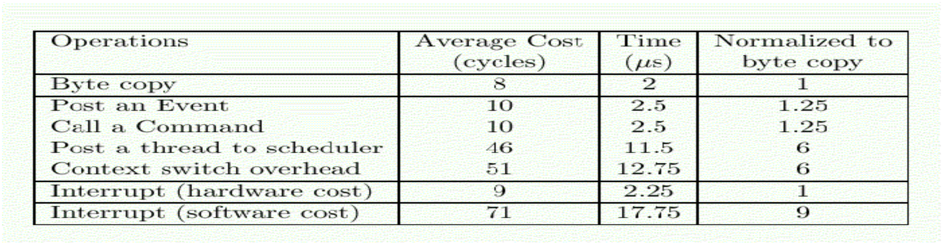
Component Name	Code Size (bytes)	Data Size (bytes)
Multihop router	88	0
AM_dispatch	40	0
AM_temperature	78	32
AM_light	146	8
AM	356	40
Packet	334	40
RADIO_byte	810	8
RFM	310	1
Photo	84	1
Temperature	64	1
UART	196	1
UART_packet	314	40
I2C_bus	198	8
Procesor_init	172	30
TinyOS scheduler	178	16
C runtime	82	0
Total	3450	226

Figure 4: Code and Data Size for TinyOS and an application [1]

- **Concurrency-Intensive Operations**

  Network Sensors need to handle multiple flows of information simultaneously. An important characteristic is the *context switch speed*. The table below shows this aspect when compared to the hardware cost for moving bytes in memory.

  The *cost of propagating an event* is roughly equivalent to that of *copying one byte* of data. *Posting a thread and switching context costs* about as much as *moving 6 bytes* of memory. Hence the TinyOS supports concurrency intensive operations effectively due to reduced context switch time.



Operations	Average Cost (cycles)	Time (μs)	Normalized to byte copy
Byte copy	8	2	1
Post an Event	10	2.5	1.25
Call a Command	10	2.5	1.25
Post a thread to scheduler	46	11.5	6
Context switch overhead	51	12.75	6
Interrupt (hardware cost)	9	2.25	1
Interrupt (software cost)	71	17.75	9

Figure 5: Cost of Primitive operations in TinyOS [3]

- **Efficient Modularity**

  The events and commands propagate through the TinyOS components very quickly. The event model triggers events quickly and commands are executed in real-time. Since the context switch time is very less and the TinyOS active messages do not waste time in copying data, a good response time is achieved. [1]

- **Communications Model Evaluation**

  The performance of the Active message model can be evaluated by using *Round Trip Time (RTT) and throughput.* [2]

  The *RTT* measures the time for a message to be sent from a Host PC to a specific sensor device and back. The RTT is plotted for various route lengths. A route length of one measures the Host-PC to base station RTT and is about *40ms*. This reflects the cost of wired link, device processing and Host OS overhead. 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*.These RTT measures indicate that the Tiny Active message Model is really fast. [2]

  Since the RTT is very less, the throughput or the messages handled in unit time is more.

6.2 Comparison of TinyOS with other Embedded OS

Comparison of *TinyOS* with common Desktop and Server OS like *MS-Windows, Sun Solaris, UNIX or IBM's AIX* is not meaningful as their application environments are totally different. These Desktop OS are meant for a *broad range* of applications and really not suited for small-embedded devices, whereas TinyOS is suited only for Networked Sensors that are embedded in a Data collection Network.

However we can compare *TinyOS* with some of the real time operating systems like *VxWorks, WinCE, PalmOS and QNX* [1] that are also meant for embedded devices. Many of these are based on *Micro kernels* that allow for capabilities to be added or removed based on system needs. These systems provide *memory protection* and *fault isolation* features that TinyOS doesn't provide. *Security* of applications is very important in larger commercial systems. Tiny OS design does not incorporate security features at all. But still security may not be that important an issue in Data collection Networks and situation monitoring.

TinyOS *does not* guarantee 100% packet delivery, as it has no time-out mechanism and receipt acknowledgement features. It is found that about *5%* of the bytes received were *corrupted* even after some error correction. Hence some newer *error correction* scheme with CRC check is required. [16]

TinyOS does very well on *Context Switch time*. It is about *12.75 µsec* whereas a QNX context switch takes about *7.3 msec*. TinyOS does well on Software footprint also as it requires only *2.16 Kbytes* whereas *VxWorks'* memory footprint is in the *hundreds of Kilobytes*. [1]

There is also a collection of smaller real time OS like *Creem, pOSEK and Ariel* that are minimal OS designed for deeply embedded systems such as motor controllers or Microwave ovens. They also have severely constrained storage and execution models. But their models tend to be *Control Centric* that is controlling access to hardware resources as opposed to TinyOS's *Data flow centric* approach. Even the pOSEK, that meets TinyOS's memory requirements, *exceeds the context switch* limitations and hence cannot meet real-time requirements. There is *no preemption* in Creem and this totally prevents real-time processing [1]

Most of these OS are based on a *Thread based Model* and these systems need to reserve additional *storage* for every thread created. Though there might be better separation of work using threads, the *storage penalty* is too much. On the other hand TinyOS is an *event* based model and because of good buffer management, it does well on storage constraints.

The TinyOS's *Active message* model helps a lot in the reduction of power consumption. Sensors can switch to a power save mode when they are not active and events would trigger them to come back into normal-operational mode. The other threaded models have to keep *polling* for some event to occur. This results in considerable power consumption. Another advantage of using events is that *polling based I/O* mechanisms see significant *performance degradation* when the number of interfaces that must be periodically checked increases.

A traditional *socket* based *TCP/IP* communication model (used by MS-Windows and Unix) is not optimal for the Networked Sensors. First of all the use of a socket model forces the system 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. The overhead associated with context switches and the storage of inactive execution contexts is too much in the case of these socket models. [3]

Secondly, the communication is *extremely expensive* for network sensors and it is advantageous to transmit as few bits as possible. In TCP/IP and UDP, there are different fields that come as an overhead like sequence numbers, addresses, port numbers, protocol types etc. A single TCP/IP packet has an *overhead of 48 Bytes*. [3]

Finally the TCP/IP protocol has a lot of overhead in 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. *Creation of intermediate copies* & *data fragmentation* proves too costly for the sensors. [3]

There is an assortment of OS such as *VxWorks, OS-9, PalmOS and QNX* that provide TCP/IP based network connectivity to embedded devices. However, these real time OS consume significantly more resources than that are currently available on the class of hardware that TinyOS works with.

Small Devices like *Palm Pilots* and *PDA's* (using PalmOS) are optimized for user response times. They have quick periods of very high activity and long periods of idle time. Networked Sensor regimes have long periods of constant data collection. [4]

The *Wireless Application Protocol (WAP)* addresses many of the same wireless device issues presented in this paper (e.g. power and CPU constraints). However, WAP is targeted mainly at *client-server* type applications. Networked Sensor domain has small autonomous devices that may operate in large numbers. [4]

But TinyOS caters to a very *small range of applications* and *hardware platforms*. It was mainly built for Embedded Networked Sensors where applications generally perform monitoring of some specific events, data collection and forwarding to a Centralized point. The event-based model using Active Messages may not be really suitable for other traditional computing environments.

6.3 Commercial Applications of TinyOS and Future Research Directions

The various **applications** of these Networked Sensors and Tiny OS are

1) Personnel Tracking and information distribution
2) Monitoring of Real-time environments and Data collection like Temperature, light, pressure etc.
3) Secure Messaging that requires trusted communication to bases using RC5 cryptography. [9]
4) Studying Life Science patterns such as Bird's Migration and retrieving ecological parameters like toxic contents in a river. [12]
5) Monitoring Enemy targets and other targets of importance

Crossbow Inc. and UC Berkeley's Computer Science Department are commercializing microsensor Motes, that help in detecting and monitoring a wide variety of targets such as an enemy personnel or chemical threats. [14] [15] TinyOS would be used in these Motes. Crossbow manufactures and sells the Networked Sensor hardware using TinyOS.

The department of Computer Science, UC-Berkeley released a new version of *TinyOS 0.6* on January 31, 2002[10]. It can be installed over Windows 2000 and Red Hat Linux platforms.

Intel has opened a new R&D laboratory in Berkeley, California that focuses on *Pro-active computing* technologies. This includes the Mote project and further development of TinyOS. [13]

Some of the Future Research Works in TinyOS are

1) Development of a better MAC layer that fits the requirements of Network Sensors. [11]
2) Incorporating Security features in Data Transmission using RC5 cryptography and some form of Memory protection schemes. [9]
3) Determine all possible limitations of TinyOS [8]
4) Incorporate TinyOS to newer Hardware Architectures [9] [12]
5) Develop techniques to deliver data more reliably and reduce data corruption.

7. 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. Efficiency and low energy use and modularity is taking precedence over FLOPS and throughput.

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. 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.

By adopting *a non-blocking*, event-driven approach, TinyOS avoids 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.

The Tiny *Active Message* programming model has reduced the storage overhead and uses the event driven message transfer to *conserve power consumption*. The adaptive transmission control scheme and the application level forwarding of multihop traffic are valuable additions. TinyOS allows users to generate highly efficient applications targeted at the emerging paradigm of networked sensors.

8. References

[1] 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.

[2] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. "A Network-Centric Approach to Embedded Software for Tiny Devices". In Proceedings of the International Workshop on Embedded Systems (EMSOFT) 2001: Tahoe City, CA, USA, October 2001

[3] Jason Hill. "A Software Architecture Supporting Networked Sensors". Masters thesis submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2000

[4] Philip Buonadonna, Jason Hill, David Culler. "Active Message Communication for Tiny Networked Sensors". In Proceedings of the IEEE conference Infocom 2001, Anchorage, Alaska, April 2001

[5] The Official TinyOS Project Website at University of California, Berkeley. http://tinyos.millennium.berkeley.edu/

[6] Presentation Slides on "A System Architecture for Networked Sensors" http://tinyos.millennium.berkeley.edu/presentations/ASPLOS_2000.ppt

[7] Presentation Slides on " How to use TinyOS" http://tinyos.millennium.berkeley.edu/presentations/TinyOS.ppt

[8] Presentation Slides on " Towards System Architecture for Tiny Networked Devices" http://tinyos.millennium.berkeley.edu/presentations/TinyOSTalk.ppt

[9] Presentation Slides on " TinyOS – Communication and Computation at the extremes" http://tinyos.millennium.berkeley.edu/presentations/Ninja_Retreat_highlight_2001.ppt

[10] The TinyOS Software Website. http://webs.cs.berkeley.edu/tos/

[11] The Abstract Web-page on TinyOS: Operating System for Sensor Networks http://buffy.eecs.berkeley.edu/IRO/Summary/01abstracts/szewczyk.1.html

[12] A News Article Web-Page on the "Daily Illini Online Magazine". http://www.dailyillini.com/oct00/oct16/news/campus02.shtml

[13] A News Article Web-Page on "Silicon Strategies Online Magazine" http://www.siliconstrategies.com/story/OEG20011112S0077

[14] A News Article Web-Page on "Sensor Mag Online" http://www.sensorsmag.com/articles/0102/10/main.shtml

[15] A News Article Web-Page on "Sensor Mag Online" http://www.sensorsmag.com/express/hardware/2001-12-07-6139.phtml

[16] Scott Klemmer, Sarah Waterson, and Kamin Whitehouse. "An Empirical Analysis of TinyOS RF Networking (and Beyond…)" . A University project Report. http://www.cs.berkeley.edu/~kubitron/courses/cs252-F00/projects/reports/project6_report.pdf