A Biologically-Inspired Approach to Designing Wireless Sensor Networks

Matthew Britton, Venus Shum, Lionel Sacks and Hamed Haddadi

Abstract—In this paper, we contend that there are significant advantages in treating some classes of sensor networks as biological-like systems—both in structural design characteristics and in operational processes. We show how this design process leads to a sensor network system that is robust to topological changes, is scaleable and self-organising—and has a number of other desirable features. The operating system kOS was designed to support the operation of distributed biologically-inspired algorithms, in order to accomplish tasks in a sensor network system. We look at the design of kOS and analyse its performance. The work presented has been implemented in an environmental monitoring project, and has applications in other areas.

Index Terms—Biological algorithms, operating systems, wireless sensor networks.

I. Introduction

Biology has inspired people in various fields to design systems in different, often elegant ways. Biological concepts have been used to inspire approaches such as artificial neural networks and evolutionary computation, and have led to alternative methods in fields such as operations research and robotics. In this paper we examine the characteristics of biological automata—simple biological "agents" which interact with their neighbours via simple rules, and yet seem to cooperate with a large number of individuals to perform some complex global task.

There are significant advantages in treating some classes of sensor networks as biological automata-like systems—both in structural design characteristics and in operational processes. Biological automata have a number of desirable characteristics such as scalability, robustness, simplicity and self-organisation—which are also desirable characteristics of wireless sensor network systems. In Section II we look at the characteristics of biological automaton systems and their applicability to the design of sensor networks.

In Sections III and IV we look at the requirements of the network and of individual nodes in more detail, and begin to assign functions which satisfy these requirements. We describe how we may develop an architecture combining both a bottom-up and top-down approach. This is done by combining a set of biological automata-like features with the top-down design of a self-organising system.

In Sections VI and VII we describe the design of *kOS*, an operating system that utilises many of these concepts to enable a network of computing devices to act like a system of biological automata.

In section VIII we describe an implementation of the kOS in the SECOAS project [24] as part of a system for oceanographic environmental monitoring [1][23]. The SECOAS project's aim is to demonstrate alternatives to the current practices of using large, expensive devices to log data—an approach that has substantial risks of platform destruction and only monitors one spatial point.

In Section IX we present some performance analysis results using the SECOAS project as a baseline, in order that our approach may be compared with others.

II. Biological Systems and Sensor Networks

A. Biological Automata

In many biological *hive* or *collective intelligence* systems such as termite colonies [27], bacterial colonies [26] and firefly swarms [28], simple neighbour-neighbour interactions between organisms or cells results in complex behaviour, where members seem to work together to perform a global task. This property is sometimes referred to as *emergence*, as significant complexity is produced from seemingly simple rules and interactions. These systems all contain distributed computational units which respond to environmental stimuli, with little or no centralised management—these units are commonly referred to as *biological automata* or agents. A common theme amongst these systems is that a change in behaviour of individual units (and

Manuscript received September 13th, 2004. This work was supported by the University College London and the UK Department of Trade and Industry (DTI).

M. Britton is with the University College London, London, WC1E7JE UK (phone: +44-20-7679-5755; e-mail: mbritton@ee.ucl.ac.uk).

eventually of the whole system) occurs when an environmental parameter is perceived to cross some threshold. Thresholds, depending on the system, may be the distance to a closest neighbour or the concentration of a chemical. From this set of characteristics, a number of remarkable traits emerge.

Firstly, the systems tend to *self-organise* and self-optimise. These systems do not rely on central management—they exhibit global optimisation of various processes by simple neighbour-to-neighbour interaction only. This is conducted via chemical or electrical media, and this diffusion or transmission process has a limited range.

As a consequence of having no central management, these systems often exhibit robustness to individual failures and topological changes. Adaptation is a major strength of biological systems, as they must response to addition or removal of members, as well as sudden changes in the environment. Because these systems have no central management and only use neighbour-neighbour interactions, they have remarkable scaling properties.

Biological systems adapt to dynamic environments, as they encounter extreme variations of various parameters in their environment—such as temperature, humidity, pressure and availability of food (energy) sources. They also show adaptation to changing requirements, as new tasks must be executed in order to counter the change in environment. Some species of ants, for example, use a simple control mechanism to maintain a constant proportion of worker to warrior castes [29].

Finally, these systems gracefully adapt to achieve a result in an iterative-like process and tend to ignore outlier (possibly erroneous) results—such as in bird flocking [27]. This is a natural consequence of using neighbour-only interactions, as automata adapt using information from a number of neighbours. This means, however, that changes are slow to spread through such a network.

B. *Application to Sensor Network Systems*

Researchers in fields such as Artificial Intelligence and Robotics [29] and Computer Networks [30] have sought to mimic various aspects of such networks of biological “automata”. Our work has shown another application for such systems. What is striking about the characteristics of these biological systems is their similarity to the requirements of sensor networks—not only in terms of structure but also the processes required to achieve global tasks. In this section we explain how the design and operation of a sensor network may naturally be viewed as a biological-like system of automata.

Wireless sensor network systems by their very nature are distributed. This is often due to the requirement of measuring a spatial field of parameters that a single sensor (even if somehow mobile) would not satisfy. In this case, centralised management is costly (in terms of power usage) due to communication and also limits the scalability of the network. It is natural, therefore, to limit communications to short range—possibly to neighbour-neighbour interactions only. This *de-centralised management* paradigm aligns well with the structure of biological automaton systems.

As a general rule in the field of sensor networks, it is desirable to limit the processing power of individual sensor nodes. This is because microcontroller units (MCUs) with limited memory and processing power have recently become exceedingly cheap. Even though these MCUs are limited in their ability, a network of such units can act as an extremely powerful system. This approach fits nicely with the characteristics of networks of biological automata and is economically attractive compared with alternative solutions.

Scalability is a key issue in the design of sensor networks. It is important, especially in environmental monitoring, because the size of the spatial field of interest will generally be unknown at the design phase. It is precisely by leveraging these characteristics from the biological world that we are able to design scaleable sensor networks.

For simplicity of management (and eventually for lowering maintenance costs) an *autonomous* system is desirable. Ideally, only occasional high-level policies are used to manage the system. Biological networks, having no central management, are candidates for this model. Extending this, we would ideally like the system to be self-organising and self-optimised. This characteristic also makes the system robust against individual nodes failure.

We would also like our system to adapt to *dynamic environments and requirements*. In environmental monitoring, various temporal phases of operation will exist—certain global tasks will be executed for monitoring ocean storms, for example—with different requirements for network traffic and node processing. Just as the size of the required network will be unknown at the design phase, so will the details of the phenomena encountered. An environmental scientist (as the system user) would likely want the system to operate in new ways after discovering and new phenomena.

Using *iterative applications* gives us many advantages. Firstly, the applications may adapt slowly to environmental and system conditions, just as their biological counterparts would do. They are also easy to manage as part of an overall system. This is because we may simply adjust their periodicity to alter the quality of their result: to adapt to radio bandwidth constraints or system processing demands, for example. When applications are constructed this way, their operation becomes simple and predictable.

As can be seen, then, virtually every one of the desirable characteristics of sensor networks has an analog from biological automata. It seems possible, therefore, to use the best ideas from the biological world and design a sensor network with a number of useful properties. One of the disadvantages of this approach, it must be noted, is that it is suitable for relatively high-latency requirement systems only. This is a consequence of having slow, iterative, neighbour-neighbour interactions between nodes. However, there remain many more advantages than disadvantages. In summary, then, our sensor network should exhibit the following characteristics:

- Decentralised management
- Self-organisation and autonomy
- Robustness to topological change
- Limited processing power of individual nodes
- Power control for individual nodes
- Adaptation to dynamic environments and changing roles

III. System Requirements

There are many tasks need to be achieved in the sensor system. We refer to this as the *functional plane* of the network. In this section, we describe what functions need to be supported across the network.

With the constraint of slow, iterative neighbour-neighbour interactions, our system operates with *weak consistency* across the network. This means that nodes do not generally need to synchronise their databases, variables, etc in order to perform their tasks. This also means that nodes may execute the same tasks at different times. Not surprisingly, biological systems exhibit the same characteristic.

However, the system does require *coordination* between nodes to some degree, where individual nodes cooperate to achieve local tasks. This is for a number of reasons. Firstly (and particularly with environmental monitoring), areas of interest will generally appear in the spatial structure of the parameter of interest. It is useful, therefore, for nodes within this area to interact and understand the phenomenon as a sub-group within the network. Secondly, by coordinating their actions, nodes may save energy by electing representatives to report on this phenomenon. Therefore the system requires some mechanism for coordination across the network. Fig. 1 shows how disparate sets of nodes contribute to various global tasks.

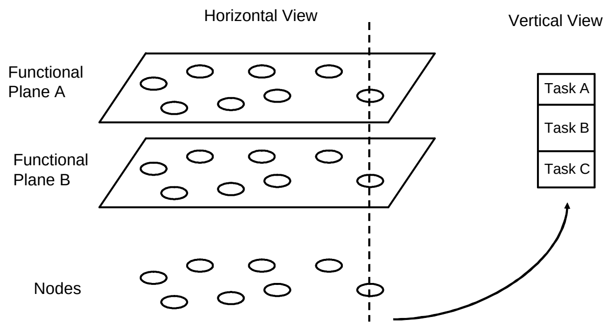

Fig. 1. Two views of the sensor network system: (a) the "horizontal" view showing layers of network functions (functional planes) upon a network of nodes, and (b) the "vertical" view functions (tasks) within one of these nodes.

As the network topology is unknown at the design stage (and may even be dynamic) we require that the data transport protocol be independent of the topology as much as possible. We also note that as nodes are expendable we wish them to be anonymous—that is, their identity is unimportant, with location being their only useful identification. As the system should be extremely scaleable, building routing tables becomes infeasible. This naturally leads to using gossip-like protocols [6][20][32] for data dissemination, as they offer a slow, reliable service for any topology. This is an acceptable method for application parameter-sharing or for policy dispersion. For data acquisition, however, we wish to use at least some directional information. This “directed gossip” protocol would have an awareness of the direction of data sinks and data would naturally forward in that general direction. Concepts such as Directional Diffusion [31] are useful for these purposes.

Power management at a system level is important as we do not wish individual nodes to fail at important junctions in the topology. We do this primarily by clustering nodes based on spatial data similarity [6][22]. These clusters nominate a node to represent them, which sends data to the sensor data source on the cluster’s behalf. In this way, excessive multi-hop radio communication is avoided and the system lifetime is extended.

Finally, we require the system to have high *integrity* operation. Integrity, in this context, means that a system behaves as expected under a wide range of operational circumstances which include system failures as well as extremes of environment. Traditionally, engineers (for example in telecommunications) have designed systems by defining a precise performance for all circumstances.

The result of this has often been expected operation, followed by catastrophic failure at some point as soon as the system or the environment has breached the bounds of its operational parameters. To avoided this, the design presented here requires that all elements, be they nodes or algorithms, of the system can adapt to failures, corrupted data or imprecision's in parameters; and still function sufficiently.

IV. Node Requirements

In this section we look at what features individual nodes require in order to support the concepts mentioned in Section III, and progress to describe requirements elicited for other reasons.

Firstly, each node requires some kind of *radio communications device* and a *sensor device* in order to measure physical parameters of interest. In this paper we treat these as two external modules, as our interest is primarily in the software layer. Also, (as shall be seen later) our implementation naturally divides each node into three separate modules with simple interfaces between them—in fact many of the features we discuss require this to be the case. Our implementation work is focussed on the “*processing module*”, and our discussion in this paper shall be mostly limited to this module unless stated otherwise.

On this processor module, tasks will need to perform simple mathematical calculations, as statistical data from sensors and other management parameters must be analysed and manipulated. As our applications generally execute in an iterative manner, the nodes require access to timers in order to schedule themselves for execution—these timers must be adjustable to enable to nodes to adapt their behaviour. Various other capabilities are required such as memory buffers for storing parameters and history states. Moreover, it is necessary to implement these node functions on a simple *microcontroller unit* (MCU). We look at the allocation of these functions to a hardware implementation more closely in Section VIII.

Power minimisation is also desirable, in order to extend the life of the network as much as possible. To do this we implement sleep or idle cycles—another standard feature in most MCU designs. We also choose an MCU which naturally has low power usage. However, with most wireless sensor networks the majority of power is expended in the radio communications device—so our design of the processor module is only a small part of the node’s power management.

As we have stated above, individual nodes perform one piece of a global task. There are multiple tasks executed in the system, such as data retrieval and various kinds of collaboration. Individuals therefore have multiple pieces of work to achieve in order to contribute to these multiple tasks.

To give each node the required flexibility to manage multiple tasks (where task schedules will generally change over time), it is necessary to have some kind of *operating system* to be executing on each node. We shall now refer to the sub-set of tasks which perform some kind of algorithm as *applications*, as they are modular pieces of program code. An operating system is also necessary in order to manage various tasks priorities, such as high-priority interrupting tasks for interfacing to an external peripheral such as a radio communication module. It is also convenient for organising applications as programs and interfacing timer devices, for example. Fig. 2 shows the logical components in each node which we have discussed to this point.

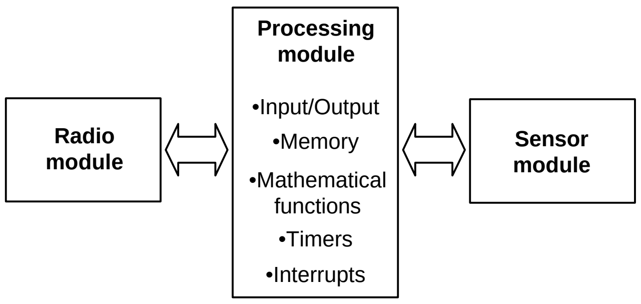

Fig. 2. The logical components (modules) within each node, showing the minimum functions needed for the processing module.

A. *Operating System Requirements*

We call this operating system “kOS”, meaning a “kind-of” or “kilobit” operating system—reflecting its relatively simple nature. The responsibility of kOS, then, is to provide a platform for the support of these distributed applications. As the applications are relatively simple in nature, the kOS does not need to support sophisticated libraries of mathematical functions or provide a capability for multi-tasking execution.

One of the first things we see, therefore, is that a simple single-tasking, run-to-completion execution model is sufficient. This greatly simplifies the design and operation of the kOS. Applications have low requirements for latency, meaning we can afford to do this without affecting their performance. The only real-time service we require is to read information from external peripherals when it is available with low latency—something most MCUs provide via an interrupt service.

The architecture of these applications depends on similar code elements (algorithms, applications) running on each node, which communicate with each other on a nearest neighbour basis. Thus, the requirement on kOS is to be highly communication oriented. To do this, we require an efficient messaging interface to be built into the kOS kernel. Applications may then use a simple call to send a message to remote nodes or to other algorithms on the same node. This design also has the advantage of decoupling application functions from their hardware implementation location. The advantage of this shall be explained in more detail later—for now we simply observe that applications may execute in other kinds of devices and still communicate using this messaging service.

To maintain simplicity and robustness, we have designed kOS to be as *stateless* as possible. kOS is stateless in at least two senses. Firstly, the kOS does not need to maintain any context information during application execution, such as stack data—as all application instances run to completion before other applications execute. Secondly, the messaging service simply informs applications of the arrival of new data, and it is their responsibility to deal with this data appropriately within a given time. Circular buffers are used so that if new data is not dealt with in time, they are overwritten by newly-arriving data. In this way, no data-application state needs to be saved by kOS and the messaging process is loosely-coupled to application execution. This is not to be confused with the state of *applications*; applications do require some small state information (such as a nearest-neighbour list for a clustering application)—our design simply assumes that kOS does not need to be aware of this.

The kOS needs to be able to manage a schedule of the various tasks on each node. For simplicity, we force repetitive tasks to self-invoke their next execution time. Because the applications are “iterative” (that is, they are constantly refining their result similar to biological automata), we can adjust their period of execution (and hence the quality of their result) and scale the processor load accordingly. This means, for example, that a node can adjust to low power or low node-node bandwidth operation by only executing tasks occasionally.

These requirements make the adoption of an object oriented paradigm appropriate. In this view, applications consist of object instances. These objects communicate through messaging—with the messages either being sent locally or over the radio communications channels. This approach provides a clear applications model which is naturally oriented towards supporting distributed applications.

As can be seen above, *power awareness* is not only a hardware issue but an operating system design issue as well. The OS has control over when the MCU should idle or sleep, and largely controls its processor load. These features allow the node to adjust for low-battery conditions, allowing the sensor system to gracefully reach the end of its (battery power) life by extending the lifetime of particular node—such as nodes at traffic bottle-necks, for example. Some of our work has been dedicated to designing distributed control applications to adjust nodes to these conditions [5].

In summary, the design of kOS is built on several guiding principles:

- Modularity of application design—via a simple messaging interface,
- Simple execution model—eliminating most context information and providing robust and predictable operation,
- Power awareness—using adaptive scheduling, as well as other standard techniques such as using shutdown features, and
- Simple processing load control—by adjusting the execution periods of iterative applications.

V. Related Work

In this section we examine the field of embedded operating systems, and compare their capabilities against our requirements stated in Section IV.A.

There are several efforts of research in embedded operating systems for wireless sensor networks. These include the Smart Dust concept [8] initiated by the Electrical Engineering and Computer Science Department at the University of California, Berkeley. This idea was later instantiated as the TinyOS operating system and the design and construction of several types of small platform “motes” able to execute TinyOS and applications [9]. TinyOS is a small footprint (at a minimum, 4kB in ROM), event-based operating environment designed for use with embedded networked sensors. Its design philosophy is complementary to ours in several ways.

Firstly, TinyOS was designed to support concurrency-intensive operations in order that multiple applications and a radio slave device (which have no buffering and strict hard real-time constraints) be multi-threaded to give acceptable levels of service. This differs from our approach of single task execution: our applications have more relaxed latency requirements and can be pre-empted by higher priority tasks with no affect on their quality. Our radio (and other external hardware devices) has its own buffering and therefore more relaxed timing constraints.

Secondly, there is a fundamental difference in philosophy between the two approaches. TinyOS was designed to be as minimal as possible in order to execute on extremely limited devices—limited in terms of size, power and processing capability. The stated goal of supporting cubic millimetre scale devices with minimal processing (the emphasis is on forwarding to an intelligent node) is vastly different from our goal: ours is to execute a small, simple OS on a cheap (possibly) wallet-sized device. Put simply, our emphasis is on the support of networked applications rather than hardware. TinyOS has been implemented in many projects; perhaps the most relevant example is its use in 2002/2003 for monitoring animal habitats on Great Duck Island, Maine, USA [10].

The EYES project, and its associated EYES operating system [19] is much closer to our goals in that it seeks to support a self-organising sensor network of distributed applications. Another related project is within the Swedish Institute of Technology; where an operating system called Contiki [25] has been designed. In Contiki, reprogrammability is an important feature which has been built into the kernel library. As with TinyOS, however, its execution model is unsuitable for our purposes.

There are various embedded operating systems that we considered. All the operating systems fail to meet our requirements because of one or more of the following: (i) they are specific to various embedded hardware functions or are minimal *executives* providing static task executions suitable only for particular applications (and do not easily allow application extensions or have only static task allocations), such as pOSEK [11] and CREEM [12]; (ii) they use multi-tasking techniques [25][9], such as POSIX-compatible threading [13], whereas we require a simple single tasking execution model; (iii) their memory footprint is excessive for our MCU-based system, such as µClinux [14]; or (iv) they execute only on powerful Pentiums or ARM processors, such as VxWorks [15] and Ariel [16].

There are also various projects similar to SECOAS, where nodes have been built and tested. Many of these projects implement a traditional communications stack—much too heavy for our purposes, as will be explained later. These include projects in the realm of “ad hoc networks” where routing protocols have been proposed [17][18] (such as to the MANET and IETF working groups), or relatively heavyweight access schemes such as Bluetooth or 802.11 have been used. Our project requirements are, rather, for a slowly-changing topology, gossip-based communications transport over a minimal communications stack and the inter-changeability of node roles. Hence we have little need for network addressing or routing.

VI. kOS Structure

In this section we examine how the requirements from Section IV.A are implemented in our operating system. We look at the components that make up the system and what their interfaces are. In Section VII we look at the operation of the kOS.

In the kOS structure, system functionality is abstracted into *objects* and *methods*. The architecture is devised in this way in order to promote a modular structure to the operating system and to allow loosely-coupled code elements to inter-operate. There are four types of objects:

- Local application objects—such as clustering or auto-location.
- Local system objects—such as the message handler and the scheduler.
- External devices such the radio and sensors are presented to application and system objects as *interface* objects. These objects exist purely to directly access the hardware required to enable external access—such as the USART device on an MCU. For example, an application object will use the message handler object (a local system object) to send a message over the radio. The message handler object will use the radio object to physically send the message to another node.
- Virtual applications which reside primarily on other devices—such as the sensor module, but whose messages must be interpreted locally. These objects are “virtual” in the sense that all that is required is that the incoming messages be handled appropriately. Generally, little processing is done and so only a small sub-set of the remote application’s code is needed. An example of a virtual object may be a virtual “networking” object. Whilst some code may reside on the kOS MCU, the bulk of the whole system’s networking code will logically be on a separate radio MCU. The virtual object may be referenced when a messaging packet contains useful network information for a kOS application; such as a received signal strength indicator useful for an auto-location application. It is useful to separate some seemingly external functions in this way because (for example) networking specifications may change during development.

The next level of abstraction is the *method*. All actions (*tasks*) undertaken by objects are the result of a method, which is an executable and referenced portion of the object code. There are a range of methods which objects may execute. Some methods are shared—for test and reset commands, for example—and some are application-specific. Fig. 3 shows how the four types of objects interact with the two types of methods. We shall examine the operation of kOS using methods in Section VII.

VII. kOS Operation

In this section we examine the two most important aspects of the kOS operation—task scheduling and messaging—and also look at other features which were elicited in earlier sections. In Section VIII we will then examine how the kOS has been implemented.

The basic operation of the kOS revolves around a sleep/activity/sleep cycle—important for power conservation. The *scheduler object* manages these sleep-wake transitions and the order of execution tasking.

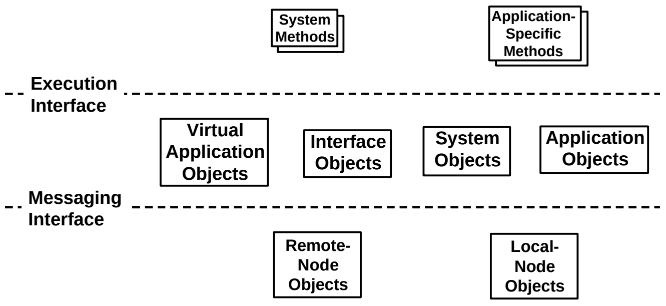

Fig. 3. The kOS functional abstraction, showing the four types of objects and two types of methods. The objects use various methods across the Execution Interface and interact with other objects across the Messaging Interface.

A. Task scheduling

The scheduler object manages the execution of object methods, or *tasks*. When a method has finished execution, the scheduler will preset a time for the next object method execution. As stated before, each application then nominates a future execute time. This basic operation is shown in a simple state diagram in Fig. 4.

As stated in previous sections, we use the biological automaton characteristic of *iteration* to design applications. This means that execution times are relatively constant each instance. Thus the scheduler can control the quality of the result an application produces simply by altering the period of its execution. We may want to do this to reduce bandwidth across our node-node radio links or to reduce power consumption when the node's battery power is low. The scheduler will do this in order to keep the total processing duty ratio below a certain threshold. We use an off-line analysis to gauge the duty cycle of each object's iteration—from here it is a simple operation in order to control the sum of all duty cycles in the kOS by adjusting each application in this way.

Examples of possible object tasks include the periodic updating of neighbour lists by a clustering application, or the periodic iterative estimation of location by an auto-location application.

B. Message handling

All application objects need to communicate with (at least) their instances on radio-adjacent nodes to be most effective—that is, they are inherently *distributed applications*. For data that must be shared across the network, we have designed a custom gossiping transport protocol [6][20] using hash functions [21] and a synchronization process inspired by firefly behaviour [32]. Using this scheme, whole-network traffic naturally adapts to local changes—traffic increases when it needed (detected via hash function exchanges) and returns to normal after exchanges occur.

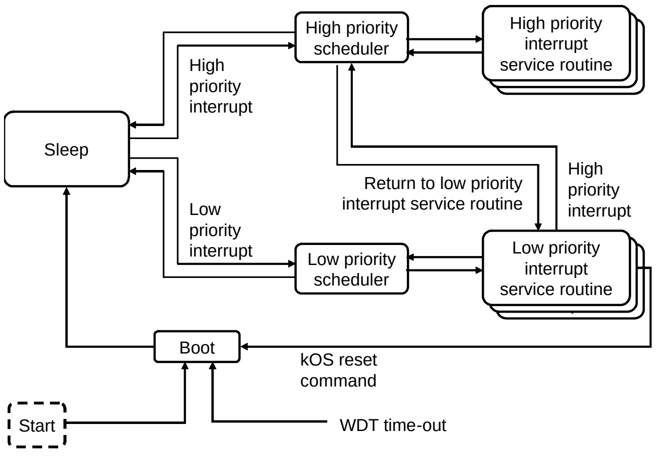

Fig. 4. Basic kOS operation, showing the sleep/wake cycle and the use of interrupt service routine for high and low-priority tasks.

The message handler object is scheduled periodically, and is additionally scheduled after any radio or sensor interface messages are received. When new data is found, the object schedules a "new data arrival" method for the destination object. When this method executes and the application sees the new data, it is the application's responsibility to read the new data and deal with it appropriately—for example, to incorporate the data into its own buffer, as the packet will be deleted due to the circular buffer operation. In this way, message handling is loosely-coupled to application execution.

A simple messaging protocol called SAM is used by objects for intra- and inter-node communication. The majority of SAM messages contain application information, such as the sharing of parameters or data.

However, any object may use SAM to pass data or control information over the radio interface to neighbouring nodes or to applications residing on the same node.

Using the message-handling service, a simple gossip-like protocol is used to disseminate various kinds of information around the network, such as policies or application parameters. For data acquisition, where we want to send sensor data to a network sink, we combine this gossip protocol with information from the specific network layer used. As we shall explain later, one possible implementation of this kind of "directed gossip" feature uses the SSSNP network service layer [2].

C. *Robustness of operation*

In our node, robustness is very important as we are operating a remote, embedded system. By *robustness*, we mean several things: (1) that each node will operate as expected, and if not, is expected to reboot itself in an attempt to bypass any intermittent problems; (2) that applications will operate given unknown radio connectivity conditions; and (3) that applications will operate acceptably when load-controlled by the scheduler (see Section A above).

The first simple step in ensuring that each node is robust in the sense of point (1) above is to implement a Watch-Dog Timer (WDT). This common MCU hardware feature simply resets the MCU if the WDT is not reset within a preset time. This is a standard method used for embedded applications. However, after a WDT power-on reset, the problem that led to the reset may occur again. As repair in impractical, we are currently investigating countering problems by ignoring (or more closely monitoring) calls to this problem objects, bypassing this problem for that particular node.

Point (2) above (unknown radio connectivity) is satisfied by designing applications in an iterative way. That is, they are called periodically so that they build the quality of their result using the latest available information during each iteration. If information from the network is unavailable for short periods of time, this simply halts the iterative process for that time period, after which the result again iterates.

Point (3) above (load-control) is also satisfied by designing "iterative" applications. This is achieved by deconstructing each application object into methods, where each method has a relatively constant execution time. It is a simple matter to change the periodicity of these applications in order to satisfy some system processing goal.

VIII. Hardware Environment

The kOS was designed to execute on a microcontroller, as it needs access to interrupts, a USART for communication, timers, logical ports, etc, and must be debugged and re-programmable. We chose Microchip Inc's PIC18F452 microcontroller [3], as it (amongst other candidates) (i) has the necessary features we wanted; (ii) was already used by one of our industrial partners and (iii) had a large user community ensuring ongoing access to timely support.

The kOS was developed using an x86-based PC, a PIC18F452 and various Microchip Inc development tools: (i) the MPLAB development environment; (ii) the C18 C-compiler, (iii) the ICD2 in -circuit debugger, and (iv) the PICDEM 2 Plus demonstration board [7]. We have chosen the PICDEM 2 board for the 1st prototype as it allows quick and cheap prototyping whilst providing the necessary basic functions we require, such as a crystal oscillator, EEPROM storage, a display and a physical interface for external communication. If we wish to produce a more sophisticated and lower-power device we may build our own circuit board in the future—at the moment our focus is to support the applications. We use two 16-bit timers on the PIC18F452s—one for tasks execution and one for a real-time reference.

A. *Node Memory Organisation*

The PIC18F452 microcontroller unit (MCU) has three main types of memory. FLASH EPROM holds the program memory, where we have the kOS object code, object methods and constants. The second memory area is the RAM—a volatile memory used for short-term data such as the high- and low-priority application queues, sensor data storage, and radio receive and transmit data buffers. The third memory area is EEPROM—a non-volatile memory used for long-term storage of data, such as a history of boot-up failures for error control. The PIC18F452 has 32k FLASH, 1.5k RAM and 200 bytes of EEPROM on-board. The PICDEM2 board provides the fourth memory area, a large (256kB) EEPROM for storage of (i) sensor data both from sensor module; (ii) sensor data from the PICDEM2 on-board temperature sensor, and (iii) data copied as it passes through the radio upstream to the base-station. This data is useful for applications—for example, to enable spatial compression of data.

B. *Power awareness*

As we have briefly stated above, we utilize the sleep function of the PIC MCU in order to put the PIC to sleep during times of inactivity. We have taken several other steps to minimize the power consumption such as replacing power voltage regulators; however there remain other simple steps we could do quickly if desired. It would be a relatively trivial step to power-down the PICDEM board while this happens—an output pin would control the activity pin on a low-power voltage regulator. During sleep, we would then power the PIC using a watch-type battery. This would drastically reduce the power requirements for the board. As we shall see in Section IX.A, there are many options for power management.

C. *Radio Network Emulation*

To test our distributed algorithms and the networking features of the kOS, we designed a peer-to-peer network emulation. Our network emulation allows any number of kOS modules to be connected via emulated radio interfaces. Boards may be set as standard nodes or base-station nodes. The network is configured via network adjacency and signal strength matrices.

D. *SECOAS Implementation*

For the SECOAS project, the external radio and sensor modules have been custom designed by our project partners. As SECOAS is an oceanographic application, we use a floating buoy containing the radio and processor boards, with a submerged tethered sensor module for analyzing environmental parameters. This configuration is shown in Fig. 5—a photograph of a buoy is shown in Fig. 6.

The radio module uses a PIC16F876 processor and a Radiometrix RX1/DX1 radio transceiver. It uses a portion of the unlicensed spectrum at 173 MHz with a Time-Division Multiple-Access (TDMA) scheme at 10 kilobits per second. To minimize power, the board only wakes for (i) radio transmissions; (ii) radio receptions and (iii) communication with the processor (kOS) board.

A networking layer called SSSNP [2] (residing in the radio board) configures which time-slots to transmit and receive in, depending on the node’s place in the topology—this allows the radio to wake-up precisely for receive time-slots. In order to conserve power, the radio board gives a small time window (20ms) for communication with the processor board. In this time, the radio and processor boards complete a handshaking operation and exchange messages, before the radio returns to sleep or its radio transmit or receive operation. The SSSNP service assigns levels in the network hierarchy which a “directed gossip” application-level transport service uses for sensor data acquisition.

The sensor module uses a PIC16F452 chip with a large flash memory storage device for sensor data. It logs pressure, turbidity, temperature and salinity and implements a distributed application [5] for adaptive sampling of these sensors, as mentioned in Section IV.A. In this way, the sensor iterates towards a sampling strategy adapted to network traffic conditions and remaining battery power. The processor and sensor modules exchange messages so that the sensor module can communicate with its node neighbours.

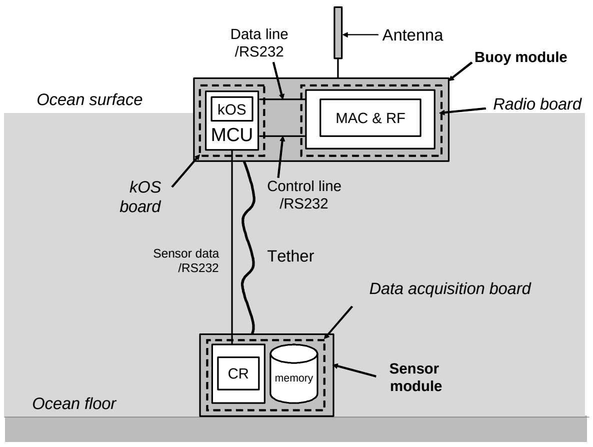

Fig. 5. SECOAS project node components showing the radio, processor (kOS) and submerged sensor module.

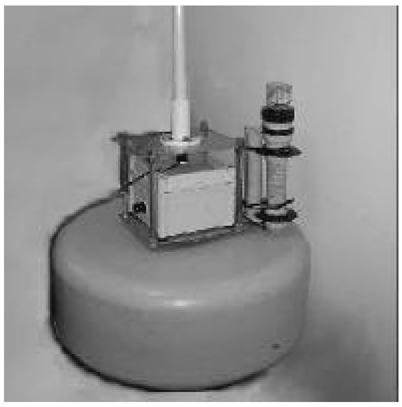

Fig. 6. Photograph of prototype buoy with mounted enclosure (white box) containing the radio and kOS modules. The antenna and navigation light are also shown. The submerged sensor module is not shown.

For SECOAS, we have also implemented a distributed auto-location application [4][22] and an application to temporally compress sensor data [22] before transmission.

IX. PERFORMANCE ANALYSIS

In this section we use the benchmark of our SECOAS project prototype design to examine the performance of kOS and its associated hardware. Note that the purpose of the prototype SECOAS design was primarily to examine the radio and network performance, together with the performance of the various distributed applications. At the time of writing, we are presently collating network results; application-specific performance analyses may be seen elsewhere [2][4][5][6][20]. In this section we restrict ourselves mostly to brief results concerning the kOS and the processing board.

A. Power usage

For the SECOAS project, we have used a 12Ah alkaline battery, which is small enough to fit into the buoy's radio/processor board enclosure. For the various elements described in Table I below, this gives a node lifetime of 299 days. This is more than adequate for our immediate testing purposes of one week, and is even adequate for the eventual goal—operation of around a year. In the current implementation, the sensor module uses a 6Ah battery—giving a lifetime of around one year.

TABLE I
CURRENT USAGE FOR VARIOUS HARDWARE ELEMENTS IN THE RADIO BUOY

Element	Current (mA)	On-duty (%)	Average current (mA)
DX1 radio transmit	9.5	2	0.19
RX1 radio receive	12	2	0.24
Radio module	1	96	0.96
LM2936 voltage regulator	0.20	1	0.002
MAX232A RS232 driver	7.90	1	0.079
kOS crystal oscillator	8.9	1	0.089
kOS PIC MCU	1.5	0.8	0.012
kOS PIC MCU sleep	0.10	99.2	0.00992
Total (mA)		**1.67**	

Note that this table does not include measurements for the sensor module.

B. CPU duty cycle

Here we show how many instruction cycles the various kOS objects use – and what the awake duty ratio is. Table II shows the list of system and application objects that are currently implemented in kOS. Note that this is specific to the SECOAS project and that execution periods can vary widely in different implementations. Note that the full duty ratio is used in the measurement in Table I. As can be seen, then, our current implementation leaves the processor board idle to around 99% of the time, extending the battery life of the node significantly.

C. Memory usage

In Table III we show how the two main types of memory (program and RAM) are used by system and application objects. The total in parentheses shows the minimal size of kOS without application objects. Our value of 12938/497 bytes compares with the TinyOS footprint of 3450/226 bytes [9]. As we explained in Section V, however, our implementation is naturally targeted toward a more capable hardware platform than TinyOS.

TABLE II
DUTY CYCLE MEASUREMENT OF KOS OBJECTS

Object	Average Instruction Cycles per second
Messenging	2335
Scheduling	4409
Gossiping	47
Clustering	1236
Data Fusion	17
Total	**8044 (0.8%)**

Note that the PIC clocked at 4 MHz operates at 1 million instructions per second.

TABLE III
MEMORY USAGE MEASUREMENTS OF KOS OBJECTS

Object	Program Size (bytes)	RAM Size (bytes)
Messenging	7928	45
Scheduling	4310	161
Data Fusion	2354	6
Gossiping	1250	39
Radio (VO)	392	128 (buffer) +35
Sensor (VO)	308	128 (buffer)
Others	3078	45
Total (bytes)	**19620 (12938)**	**587 (497)**

Note that the code size reduces by around 26% when no debugging is used.

X. CONCLUSION

In this paper we have shown how treating wireless sensor networks like biological automata can result in beneficial features such as scalability, robustness and self-organisation. A particular implementation has been shown in the kOS operating system; however, the novel work here is the design of the systems architecture. As we have shown, this novelty is primarily in they way we construct and support distributed applications—this framework could easily be built as a service layer over an alternative operating system. There are several disadvantages in constraining applications to this kind of system—however, the advantages listed above far outweigh the disadvantages for many applications.

XI. ACKNOWLEDGEMENTS

The authors would like to thank Ian Henning, Steve Fitz and Taimur Khan from the University of Essex for information on the radio module, and their assistance with designing the inter-module communication interface. We would also like to thank John Argirakis and Nathan Boyd from Intelisys Ltd for information on the sensor module.

XII. References

[1] L. Sacks, M. Britton, I. Wokoma, A. Marbini, T. Adebutu, I. Marshall, C. Roadknight, J. Tateson, D. Robinson and A. Gonzalez-Velazquez, “The development of a robust, autonomous sensor network platform for environmental monitoring,” in Sensors and their Applications XXII, Limerick, Ireland, 2nd-4th September, 2003.

[2] A. Gonzalez-Velazquez, M. Britton, L. Sacks and I. Marshall, “Energy savings in wireless ad-hoc sensor networks as a result of network synchronisation,” in the *London Communications Symposium*, University College London, 8th-9th September, 2003.

[3] PIC18FXX2 Data Sheet, Microchip Technology Inc, Document DS39564B, 2002.

[4] T. Adebutu, L. Sacks and I. Marshall, “Simple position estimation for wireless sensor networks,” in the *London Communications Symposium*, University College London, 8th-9th September, 2003.

[5] C. Roadknight and I. Marshall, “Sensor networks of intelligent devices,” In Proceedings EWSN2004, pp. 58-61, 2004.

[6] I. Wokoma, L. Sacks and I. Marshall, “Clustering in sensor networks using quorum sensing,” in the *London Communications Symposium*, University College London, 8th-9th September, 2003.

[7] PICDEM 2 Plus User’s Guide, Microchip Technology Inc, Document DS51275A, 2000.

[8] K. Pister, J. Kahn and B. Boser, “Smart Dust: wireless networks of millimeter-scale sensor nodes,” Highlight Article in 1999 Electronics Research Laboratory Research Summary, 1999.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler and K. Pister, “System architecture directions for networked sensors,” in the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.

[10] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler and J. Anderson, “Wireless sensor networks for habitat monitoring,” in the 2002 ACM International Workshop on Wireless Sensor *Networks and Applications*, September 28, 2002, Atlanta, GA, USA.

[11] pOSEK, “A super-small, scalable real-time operating system for high-volume, deeply embedded applications,” http://www.isi.com/products/posek/index.htm.

[12] B. Kauler, “CREEM Concurrent Realtime Embedded Executive for Microcontrollers,” www.goofee.com/creem.htm.

[13] S. Kleiman, D. Shah, B. Smaalders, “Programming with threads,” Prentice Hall, Mountain View, California, 1996.

[14] µCLinux Project. www.uclinux.org.

[15] Wind River web-site. www.windriver.com.

[16] Microware web-site. www.microware.com.

[17] D. Maltz, J. Broch and D. Johnson, “Experiences designing and building a multi-hop wireless ad hoc network testbed,” CMU School of Computer Science Technical Report CMU-CS-99-116, 1999.

[18] S. Desilva and S. Das, “Experimental evaluation of a wireless ad hoc network,” in Proceedings of the 9th Int. Conf. on Computer Communications and Networks (IC3N), Las Vegas, October 2000.

[19] S. Dulman and P. Havinga, “Operating system fundamentals for the EYES distributed sensor network,” Progress Report 2002, Utrecht, the Netherlands, October 2002.

[20] I. Wokoma, L. Sacks and I. Marshall, “Biologically inspired models for sensor network design,” in the *London Communications Symposium*, University College London, September, 2002.

[21] M. Castro, P. Druschel, Y. C. Hu and A. Rowstron, "Exploiting network proximity in distributed hash tables," presented at the FuDiCo 2002 International Workshop on Future Directions in Distributed Computing, Bologna, Italy, June 2002.

[22] L. Shum, I. Wokoma, T. Adebutu, A. Marbini, L. Sacks and M. Britton, “Distributed algorithm implementation and interaction in Wireless Sensor Networks,” 2nd International Workshop on Sensor and Actor Network Protocols and Applications, Boston, Aug 20th, 2004.

[23] M. Britton and L. Sacks, “The SECOAS project: development of a self organizing, wireless sensor network for environmental monitoring,” 2nd International Workshop on Sensor and Actor Network Protocols and Applications, Boston, Aug 20th, 2004.

[24] SECOAS web site. www.adastral.ucl.ac.uk/sensornets/secoas

[25] A. Dunkels, B. Grönvall and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” In Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I), Tampa, USA, November 2004.

[26] M. Miller and B. Bassler, “Quorum sensing in bacteria,” Annual Review in Microbiology, pp. 165-199, 2001.

[27] G. Flake, *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*, MIT Press, Cambridge, USA, 1999.

[28] C. Teuscher and M. Capcarrere, “On fireflies, cellular systems, and evolware,” Lecture Notes in Computer Science, Volume 2606 / 2003, pp. 1–12, August 2003.

[29] E. Bonabeau, M. Dorigo and G. Theraulaz, *Swarm Intelligence: from Natural to Artificial Systems*, Santa Fe Institute, Oxford University Press, 1999.

[30] G. Di Caro, “Two ant colony algorithms for best-effort routing in datagram networks,” 10th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS’98), 1998.

[31] C. Intanagonwiwat, R. Govindan and D. Estrin, “Directed diffusion: A scalable and robust communication paradigm for sensor networks,” in Mobile Computing and Networking, 2000.

[32] I. Wokoma I. Liabotis, O. Prnjat, L. Sacks and I. Marshall, “A weakly coupled adaptive gossip protocol for application level active networks,” POLICY 2002, 2002.