OSone: A Distributed Operating System for Energy Efficient Sensor Network

Bence Pasztor, University of Cambridge, bence.pasztor@cl.cam.ac.uk
Pan Hui, CSE Department, Hong Kong University of Science and Technology, Hong Kong,
Telekom Innovation Laboratories, Germany, CSE Department, Aalto University, Finland, panhui@cse.ust.hk

Abstract—In this paper, we take a new approach of thinking about programming Wireless Sensor Networks (WSNs) and introduce OSone, a distributed operating system (OS) for sensor transparency. Our philosophy is to make the network look like an ordinary computer, where each sensor of the network can be thought of one or multiple applications. Such a system allows software developers to abstract away from networking protocols and low-level operating system issues, and develop complex, cooperating systems. In our system, the base station acts as the "kernel" of the OS, while applications run on the sensor nodes. To evaluate the system, we use smart home as our application scenario, where the house adapts to the people living in it - turns on and off the heating and light, warns if the doors are left open, and so on. For such an application, a sensor network is the ideal solution. We show that our system can scale up to a hundred nodes without affecting the responsiveness, it can move 96% of the energy consumption to the central kernel node; and can be about 30% more efficient than a traditional approach.

I. Introduction

As a result of much research, Wireless Sensor Networks (WSNs) are starting to emerge in the real world, as opposed to being "stuck" in lab environments or network simulators. Systems are being deployed to monitor people using mobile phones [23] [14] or medical sensors [16], nature [18] or buildings [5] using static sensors, or animals using collars [10].

Up to now, every sensor network programmer had to deal with the optimisations of networking protocols and operating system to make the most out of the deployment. Years of research have made these protocols energy efficient and reliable enough for application developers to stop worrying about them, and start thinking about higher-level, complex problems. We want the application developers to think more about the applications, and less about the underlying operations. We argue that, by not having to spend time "hacking" the OS on every single sensor, developers can design more efficient, and perhaps more complex applications on the network. We propose a new way of looking at a wireless sensor network: treat it as one cooperating system, as opposed to a network of isolated devices.

We believe a sensor network is much more than a bunch of tiny, stand-alone devices logging data, and perhaps delivering them to some central storage. One can envisage these devices, instead, as part of a closely tied system, collaborating seamlessly. We want to draw the similarity between such a system and an ordinary desktop machine, running multiple, parallel processes at the same time. Such a machine is controlled by an operating system, which takes care of the inter-process communication, as well as storage or other I/O services. We envision our sensor network to work similarly: each process is "hosted" by a sensor device, while the base station acts as the "kernel" of the system. Similarly to a traditional operating system, processes could seamlessly communicate with each other (i.e. post or accept events to/from each other) and store or retrieve data from some non-volatile storage. The inter-process communication is handled by a thin layer running on the *hosts*, communicating wirelessly with the *kernel* node if necessary, without being exposed to the application. This view was motivated by the fact that in many cases, the sensors do simple tasks, such as reporting their sensor data to the sink, therefore they do not necessarily have to run a complete operating system, and thus putting extra load on their CPUs. Based on these concepts, we introduce OSone, our new distributed operating system for sensor networks.

The underlying philosophy of Osone is simple: it is much easier to manage one operating system running multiple programs, than multiple sensors networked together, each running an operating system, the applications and the numerous protocols required to provide the communication between the sensors. For large, complex systems, it also makes sense to leave the management up to an automated kernel than to have a person manually install and configure each application or sensor. Furthermore, the system perfectly suits future sensor networks, where the actual sensors have even smaller memory and computational power than today's devices - e.g. batteryless devices [24], that scavenge energy from the environment enough only for a few hundred CPU cycles.

The contribution of this paper are the following:

- OSone, a novel operating system running over multiple devices, thus simplifying the task of the application developers, and making smart use of the resources.
- A proof of concept implementation of OSone for a smart home environment, including a storage system and a TDMA-based, reliable MAC protocol. The proof of concept illustrates how easy it is to deploy traditional applications on our system.
- An evaluation of the protocol in a simulation scenario, and over a sensor network testbed.

The rest of the paper is structured as follows. In the next section, we present the main idea of the system followed by the related works in the area, while in Sec. IV we show the design and details of how our scenario is implemented on a

sensor network. In the section after, we show the details of the implementation, while in Sec. VI we evaluate the performance using simulation runs and real sensors. In Sec. VII we show future works, while we conclude the paper in Sec. VIII.

II. ARCHITECTURE

This section will present the "big picture": how we envisage the system without constraining ourselves to the limitations of any implementation or hardware.

A. Motivation

As introduced in the previous section, we would like to abstract away from the traditional way of dealing with WSN. Why would we want to do this? Simple: *system utilisation* and *efficiency*. On a traditional WSN, the nodes of the network act as stand-alone, separate entities, responsible for their own duties and managing their own resources. This requires system designers to perceive the system as a complex, distributed, sometimes even competitive set of devices, trying to work together to achieve the goal. Making such a system work requires efficient routing and dissemination protocols, complex storage systems, time management and so on. Although using efficient protocols for each subtask can achieve an energy efficient operation, it remains a complex system with lots of opportunities for bugs and faults. On the other hand, for such systems, the application itself is usually the one getting the least attention, since most of the time is actually spent on optimizing the drivers or the communication layer.

In our view, it should be the other way around. System designers should spend most of the time designing the applications (which themselves should be the complex part of the system), without worrying about the underlying system. The most obvious solution is to view the entire WSN as one computer running multiple applications. Although we are targeting a rather simple application - that of a smart home -, the general idea is much greater.

Since this section was about our grand vision, it is difficult to discuss efficiency and energy usage: these measures strongly depend on the design and implementation details. However, we argue that our system design - the *one operating system view* - gives plenty of room for resource optimisation, and thus efficiency. The section containing the details of our implementation will show such enhancements.

B. System Design

Our architecture can both be seen as a distributed system or a centralised one: distributed, since the applications may reside on different physical devices, while a centralised one since everything is controlled and managed by the kernel; just like with any ordinary operating system.

Our system does not target a specific application, but should be taken as a general platform for any sensor network. The main contribution of this paper is the idea of perceiving and handling the WSN as one system, instead of distributed networked sensors, thus opening up more opportunities for system designers. While we want to generalise to any system, we also recognize the challenges associated with different scenarios. For example, the communication layer (described in Sect. V) may require certain adjustment for different scenario, however we encourage the research community to take a step away from the distributed, inter-connected node perception towards a more system-wide view.

The system design is rather simple: it is made up of *host nodes* and a *kernel node*. The kernel node is more powerful, and has "unlimited" amount of memory, while the host nodes are more constrained. When an application is installed, the kernel node decides which host node should run the application based on the requirements of the application and the capabilities of the host nodes. All communication is done via the kernel node (be it data operations or inter-process communication). More details on the system is given in the next section. We will briefly address common issues with such systems:

Fault tolerance There are essentially two types of faults to deal with: local problems, and communication failures. Depending on the capabilities of the layer running on the host nodes, it can monitor the applications, and kill them, if needed. It can also take care of memory violations by catching all attempts of memory allocation. To keep the host nodes simple, we moved most of the fault management logic to the kernel node, therefore the most likely errors are, the ones where the sensor devices fail completely due to hardware failures, and become unresponsive to the kernel node. In such cases, the kernel node must be able to deal with the removal of the device - i.e. no messages will be transmitted to that node, and its applications will be removed from the routing tables. If the device comes back after some time, the kernel node needs to be able to re-initialise it with the correct time, and applications.

Atomic Transactions When an application runs, it runs in its own memory slot, any data in the memory used by an application should only be accessible by that application. Data on non-volatile memory is stored by the kernel. Each piece of data has an owner (an instance of an application), therefore appropriate permissions can be set.

Deadlocks and concurrency The system is truly multi-threaded - since each application is probably run on a different device, they all have a small dedicated physical CPU. Communication between applications or even with the kernel may have some delays, and thus cannot guarantee the order of messages. This needs to be taken into account when designing the application - there is, however, a method to lock a piece of data, so that no other instance of the application can access it, thus avoid inconsistencies. Furthermore, local cache is always synchronised with the kernel to keep the local copy updated.

Load balancing and resource management This issue is actually handled very well by the system - since it is the kernel which decides which application should run on which device based on the requirements of the application, it can optimise for resource usage. If, for instance, there are two devices in the vicinity, both able to perform the same task, the kernel will issue the application to the device with a higher battery capacity (or the better device, based on any metric). This is an advantage of a centralised system.

Naming and protection When an instance of an application stores some data, the data is tagged with the ID of the application instance, and thus can be retrieved easily. Ideally, the kernel keeps a filesystem, and applications can store and retrieve files, append to files, and perform other operations. In our implementation, we simply serialise the data, and send it over to the kernel as a stream of bytes. File permissions take care of data security to some extent, while the kernel can monitor the communication between the applications, and block any unpermitted channel - i.e. if an application is not allowed to talk to another.

III. RELATED WORKS

In this section, we will summaries the related works in the field of distributed operating systems, before moving onto the details of our protocol.

Research on traditional, distributed operating systems dates back to the 80's, while the first successful distributed operating system for traditional computers was introduced in the early 90's [21]. Amoeba is an object-based, client-server type of system. Each object can be seen as a process, and they are identified by capabilities ensuring protection. Objects communicate via RPCs, and the system stays hidden from application programmers. There are numerous similarities between our system and Amoeba - we also try to hide the underlying communication from the user, while providing similar functionalities to Amoeba. As Amoeba was designed for traditional systems, they put less emphasis on being energy efficient, and dealing with limited resources or bandwidth. Research did not stop at Amoeba, and the area of distributed operating systems is nicely summarised in [6].

The work of Liu et al. [15] describes a system with aims very similar to ours. They also designed a distributed operating system, where each application may run on different devices, while having a common abstraction underneath them. MagnetOS (their operating system) was implemented as a JVM for x86, Transmeta tablets, and StrongArm PocketPC devices. Although their goal was similar to ours, their system would not suit our target scenario of Wireless sensor networks due to their high resource requirements - unfortunately, their prediction of sensors having large RAM and fast CPU did not come true.

Sensor Network Services Platform (SNSP) [11] is a distributed operating system tailored for sensor networks based on a service-oriented model. Their focus is on resource allocation and content replication: given a process with its requirements, the system tries to find optimal place(s) to execute it in order to ensure reliability and availability. They do not deal with file systems, or inter-process communication.

Interestingly, there are a lot of similarities between our system and BarrelFish [3], a distributed operating system for multicore architectures. In their model, they treat the machine as a network of independent cores, where these cores communicate via message passing. Although our research domains are very different, the perspective of splitting up the operating system - while hiding this from the application - is similar.

There are several middleware approaches to a similar objective to ours. Logical Neighborhood [20] provides a way to define a set of sensors performing a certain task or running certain programs. Instead of specifically addressing the nodes, they let the user define some rules according to which the target set is selected. In some ways, this could be seen as a distributed system, however, instead of shrinking the code size, and simplifying the communication between the nodes, it introduces complexity in the underlying system.

Abstract Regions [25] could also be seen as an 'abstract operating system' — for example, an abstract region would be the set of nodes within 1 radio hops. It is then possible to share or aggregate data within a region, allowing the programmer to avoid dealing with the low-level peculiarities of the sensors. However, in our work, we provide low-level operating system functions to the applications as well as an interface to the sensors (as what Abstract Region was designed for).

In some sense, TinyDB [17] or Directed Diffusion [13] can also be seen as specialized distributed systems. Both are designed for data collection: some query is injected at the base station, efficiently disseminated to the nodes of interest, and the results are routed back to the base station. Our system is simpler than these: we propose every node to be directly connected to a *kernel* node controlling it. Had we relaxed this requirement, we would have had to introduce more complexity on the *host* nodes - against our desire to keep them as simple as possible.

IV. DISTRIBUTED OPERATING SYSTEM

In this section, we will give describe OSone, its functionalities and properties. Essentially, the goal is to provide the same functionalities as an ordinary system, on top of a wireless topology.

A. Host nodes

We call *host* nodes the small, battery-powered sensor devices equipped with some physical sensors, limited memory, limited computational capacity, and a radio. The idea is that these devices should be relatively cheap, and easy to replace or add to the system - thus the wireless connectivity and the battery power source. It is important that these devices run energy efficient protocols, otherwise the user needs to change the batteries often, which is not desirable. The main sources of power consumption on today's motes are the radio and the flash memory [19]. Transmitting a byte over the radio or storing it on some external flash memory is on the same order of magnitude in terms of energy usage (around 20 mA), therefore our aim is to optimise these operations.

The idea is to have only the very necessary software running on the sensors, thus reducing the processing, storage and communication overhead. For the system to work, there needs to be a thin layer running on the node responsible for the following functions: it needs to be able to post, catch and broadcast events between processes, as well as store and retrieve data. Moreover, it needs to run some basic drivers (CPU, radio and installed sensors) required to operate the mote normally.

To be precise, the layer provides these functions:

- $start\_process(pName, data)$ and $stop\_process(pName)$ to start or stop an application process with some optional data.
- $post(ev, pName, data)$ to post an event to a specific process with some optional data.
- $store\_data(sn, data, size)$ and $get\_data(sn, callback, size)$ are used to store a piece of data in the system as well as retrieve it.
- $get\_system\_time()$ to get the system-wide time.
- $sensors\_init(sensorType)$ to initialize and run sensor drivers installed on the mote.

To implement the above functions, the layer has two main tasks: driver control and process management. Driver control is to enable the software to communicate with the hardware: CPU, radio, memory and external sensors. The other task includes starting, stopping processes, and routing data and messages between them, as well as to the rest of the network.

B. Kernel nodes

The *kernel* node is the center of our system. We envisage this node as a more powerful device (perhaps a full computer) with infinite storage as well as computational power. This node is the injection point for new programs, controls the communication between the processes running in the system, and also implements the storage.

Software installation happens automatically, at the kernel node. When the system starts, all the external nodes start sending their service discovery messages, containing the capabilities of the *host* (such as installed sensorsm available battery, etc). The *kernel* node logs each *host* with their associated capabilities, and replies to them with a unicast message, assigning each an identifier.

Part of the installation process is to check the requirements of the application. More precisely, we are interested in what hardware features the application needs access to. If all, what the application does is aggregating incoming data, then it needs to be run on the kernel node. However, if it needs access to temperature sensor, then it needs to be run on a sensor equipped with the required part. This requires the application to define its hardware requirements. This can either be done automatically, however, for our scope, it is defined manually by the application programmer.

Once a match has been found between an application and a set of nodes, the application is delivered (as a binary image) to those nodes using controlled broadcast messages. When the nodes receive the entire image, they dynamically load it to their memory, and start the process. If the kernel node cannot find a matching sensor, an error is thrown.

Apart from application installation, the kernel node also implements the central storage. Whenever an application calls the appropriate function (store_data()), the data is sent to the kernel node, and stored - the details of our filesystem is described later on.

V. IMPLEMENTATION DETAILS

To show the feasability of out system, we implemented a simple 'smart-home' control to monitor temperature, humidity and light in a house. The system was implemented on the Tmote Sky [2] sensor board. It is a widely used development board with a simple CPU, limited amount of memory and a Zigbee-compatible radio.

Although we emphasised that the kernel node can be a more powerful device (or even a desktop machine), in our current implementation, it is also a Tmote Sky sensor node. For our scenario and application, a TMote is sufficient, however we can easily deploy it on a more powerful device, such as the iMote2 [1]. Here, the Tmote Sky case also give us a lower bound of the system performance and we can have a very realistic view of the system.

A. Process installation

As described before, the programs are installed at the kernel node. In our current implementation, the application needs to be compiled offline as a dynamically loadable ELF file[8] - Executable binary Linked Format, a standard and very flexible format to store compiled objects. This ELF file needs to be sent to the kernel node, which can examine the file, and decide which node to install it on. The applications are tagged by its requirements by the developer, and the kernel node uses these tags to find a suitable host. If it needs to be installed on a set of external nodes, the image is delivered by a series of multicast messages. If a node receives a program that it should run, it calls its elfloader function, which allocates memory, and loads the process. The elfloader is part of the Contiki operating system.

B. Central storage

We start the implementation description with the details of the central storage. As described before, all the data is stored on the central server, however, this is hidden from the applications - they simply call store_data(), and assume it to be available whenever needed. We split up the main storage into a fix-sized chunks (128 bytes), and store each piece of data in these chunks. Every piece of received data is identified by the application ID, and a sequence number sent by that application. The main difficulty is making such a system efficient over a wireless channel: we employ multiple levels of caching to enhance response time to data retrieval. The control layer running on the host nodes (which essentially first receives the data from the application) assigns the unique identifier to the stream of data, and stores it in its local cache. Once the cache becomes full or after some timeout, the data is serialised, and sent to the kernel node, where it is stored on a non-volatile storage. When an application asks for a piece of data, the local controller checks if it is in its local cache - if so, it returns the data immediately. If not, then it asks the kernel node for the data, and returns to the application when the data arrives. An error is returned if the kernel does not reply, or cannot find the data. Similarly, there is caching on the kernel nodes: each sensor node is associated with a local cache, where the last-stored data is kept, thus it can avoid spending time reading the non-volatile storage.

As storage operations are hidden from the application, the functions should return quickly: i.e. when an application

asks for a data, it should receive it rather soon. The more intuitive option is to have the application block until the data arrives, however this is not safe when dealing with wireless communication. Instead, we ask the application to set up a callback function, which is called upon the arrival of the data (either immediately from local cache, or from the kernel node). In fact, this is our bottleneck - since all the queries are served by the kernel node, some delay can be expected - shown in Sect. VI.

C. Inter-process communication

Similarly to the storage system, inter-process communication should be hidden from the application. On the sensor nodes, the control layer keeps track of what applications are running on the nodes, as well as a callback function in case the application receives an event. When an application wants to notify another running process, it can post an event. These events can either be defined by the application, or the OS. Along with the event, the application has the option to send a fixed amount of data. The event is caught by the control layer, and forwarded to any local destinations (via the registered callback), as well as sent to the kernel node, which in turn can inform any other destination on remote nodes in the system. The kernel node delivers the event notifications using a multicast message. It is a single broadcast message destined to a set of nodes. Each receiving node checks if it is part of the destination set, and acts accordingly.

D. Underlying communication

Since this was our first attempt at solving the problem, we had to make some compromises during the implementation period. One such compromise was the use of a centralised kernel, able to reach every node in one hop. Future versions may allow multi-hop deliveries.

To support the above described functions, we needed a communication stack that can provide time-guarantees. It is important to be able to guarantee a reply to, for example, a data retrieve or a process notification. Our centralised scenario allows us to use a more-controlled way, so we decided to use a TDMA scheme. The time is split up into small time-periods, and each node is assigned one. Since the nodes mostly talk to the kernel node, it will have a higher traffic demand - the scheme allows us to assign multiple slots to the kernel node, so that it can respond quickly to queries, and does not have to wait a full cycle.

For higher layer, reliable data transfers, acknowledgements are required within a set time-interval (set to 1 s), otherwise the packet will time out. We chose to use a quick cycle period for the MAC protocol: each host node gets a slot every half a second, while the kernel node has a slot after every host node. In other words, every host node has one slot within a cycle period, while the kernel node has a slot for every host node. This is necessary, since all the communication happens with the kernel node, therefore this node needs to have proportionally more chances to transmit, based on the number of host nodes. Depending on the slot length, a node may have enough time to transmit multiple messages. We also
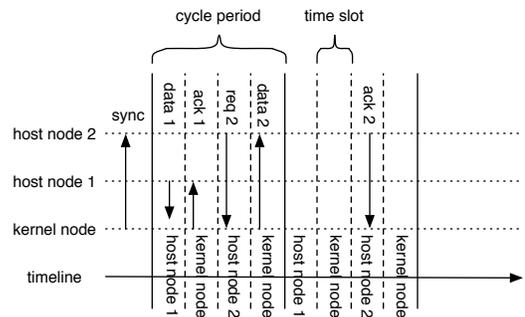

Fig. 1. Our TDMA scheme with 2 host nodes and a kernel.

use buffering on the MAC-layer, in case the node does not have enough time to transmit the packet, it can transmit it in the next cycle.

For the TDMA scheme to work across the system, the nodes need to be synchronised in the order of milliseconds. This is achieved by using a very simple, one-hop time synchronisation protocol, where all the nodes synchronise their local clock with that of the kernel node. It time-stamps every outgoing packet (on the driver level, just before writing the packet in the radio buffer) with its local clock. The receiving node, again, in the driver, just after receiving the packet, checks the time-stamp, subtracts the approximate duration of the message delivery, and compares the time-stamp to its local clock. The difference (i.e. the offset) is added to the local clock, thus accounting for any time-shifts between the kernel node and the node.

As we have a strict TDMA protocol, there is a good opportunity to duty-cycle the radio on the devices, since we know when we can expect traffic. Once the host nodes synchronised their radios with the kernel, they only turn their radios on during their own time-slots (and the consecutive time-slot to listen to possible replies from the kernel), and turn the radios off during other time-slots. This results in great energy savings (the hosts are using only 4% of the energy used by the kernel, as shown in Sect. VI) on the host side.

Fig. 1 shows an example of our TDMA scheme with 2 host nodes and a kernel node. The first message is an explicit synchronisation message broadcast to all nodes. Since there are 2 host nodes, the cycle is split into 4 slots: one-one slot to the host nodes, and the rest for the kernel node. In this example, *host node 1* sends a data message to the kernel, which then sends an acknowledgement back, while *host node 2* requests a data packet, which the kernel node delivers, and the host node then acknowledges in the next slot.

Since the nodes need to receive at least one message from the kernel node to synchronise their clocks, there might be collisions occurring until this happens.

There are three types of messages sent by the system: broadcast, multicast and unicast messages. We use Contiki's Rime communication stack [7] to implement each type. Broadcast messages are used at boot up time by the host nodes to find the kernel node. Multicast messages are sent by the kernel node to deliver application images as well as event notifications. Unicast messages are used to deliver data both to and from the kernel nodes, as well as initialise the host nodes. All

of the unicast messages require an acknowledgement from the receiver, otherwise they time out, and will be re-sent a predefined number of times.

VI. EVALUATION

A. Overview

Here we give the results of our evaluation based on real sensor motes and in a simulated setting.

For the evaluation of the system, we implemented a scenario of a *smart home*. A smart home is a sensor-equipped house, where the environment is adjusted to the current context. For example, lights turn on when a person enters the room, the heating turns to standby when everybody leaves the house, etc. As well as making the life of people living in the house easier, it also ensures environmental friendliness - the computer will not forget to turn off the lights or close the windows, thus reducing the carbon footprint of the house.

We propose the following scenario to present the capabilities of our system: imagine a smart-home with tens or perhaps hundreds of sensors deployed, all connected to a central base station. More specifically, we envisage motion detectors, temperature and light sensors throughout the house. This is a highly heterogeneous system, with sensors having different capabilities. If, now, we treat the entire network as an operating system, we can imagine the following applications or programs running: one to monitor any motion in the house, one to report the temperature and one to report the light levels, one to keep the temperature and light levels in the house at desired values, and one to run security and safety monitoring program. When running, the programs cooperate and communicate with each other to perform their associated tasks.

We evaluate the scalability and efficiency of the system using the following metrics:

- *response time* is an important metric for such systems - applications may become stuck if they do not receive a reply for a query.
- *radio dutycycle* measures the proportion of time the nodes need to keep their radios on, and thus consuming energy.
- *energy consumption* logs the energy consumed by the nodes throughout our test runs.
- *CPU usage* shows the amount of time the nodes kept their CPUs active, and how much they could put their CPUs to low-power mode.
- *memory footprint* is based on our proof-of-concept implementation on real, off-the-shelf sensor devices.

This section shows the evaluation of our implementation, as opposed to the one-operating system idea. It is difficult to quantify, on the short term, how much easier it is to design and maintain such a system compared to a traditional one.

B. Static Analysis

Along with the simulations and the real-world evaluation, here we will give some information about our implementation. As described in Section V, we implemented our system in Contiki. The goal was to shrink the code base on the *host nodes* as much as possible, and move as much as possible to the *kernel nodes*.

Our system is made up of the following components: system drivers (radio, MCU, serial line - about 3K), networking (timesynchronisation, TDMA-mac, rime communication stack - about 1.9 K) and the control layer (timers, process-control and cache - about 3 K). We used the *msp430-size* application on the compiled image to find the exact size of each component. The host node requires about 8K RAM, out of which more than 2K is the radio driver. Purely our system (control layer + TDMA-mac) uses about 4K on the host. This is quite memory efficient, and leaves a lot of room for applications. We implemented the kernel node also on the Tmote Sky platform, using almost 22K RAM. Although this is almost half of the available memory on the mote, the kernel node could be implemented on a much more powerful platform, with more memory available.

We also found the memory consumption of the same application running on the unmodified Contiki OS using its filesystem, and a simple code delivery protocol. Its code size in terms of RAM consumption is slightly less than that of the kernel node. The advantage, however is not the code size, but rather the energy usage, as shown later in this section.

C. Testbed Evaluation

Our intention from the beginning was to design something practical and useful, therefore implementing the system on real sensor motes was of high priority. To evaluate the system, we designed a simple test application for smart homes. The application queries the temperature sensor of the device every one seconds, and along with a time-stamp, tries to store it. The devices record the time it took to receive the acknowledgement from the kernel node, confirming that the storage was successful. After 10 store instructions, the sensor sends out a process notification, and again, the time to reach the other processes is logged. At the end of the test run, the 10 data points are requested from the storage system, recording the time from the request to the actual arrival of the data points.

The system was evaluated on the University of Bern Testbed using their Testbed Management solution TARWIS, part of the WISEBED project [12]. The testbed contains 20 wirelessly connected TelosB motes (equivalent to the Berkeley or TMote Sky motes). The system allows the upload of a compiled image, and the capture of the output from the motes. Since we need a centralised system, we could only use the fully-connected subset of the network, containing 14 nodes.

We also compared our system to a traditional implementation: the same test application is running on standard Contiki, and the radio is dutycycled using the X-MAC protocol [4]. It is a preamble based protocol in which senders indicate their intent to send data by frequently transmitting short wake up messages. Nodes periodically wake up, and if they hear a preamble indicating a packet is addressed to them, they respond with an acknowledgement. This terminates the wakeup phase and the packet is sent.

1) Response time: The first two functions to test are *store_data(...)* and *retrieve_data(...)*. In both cases, we logged the time the kernel node responded with an acknowledgement, and averaged these to get an idea of the responsiveness of the

system. In both cases, the response times depend on a number of parameters: the number of host nodes in the network and the local buffer size. The first parameter affects the number of slots in a cycle (and thus the slot length) while the second can reduce the amount of transmissions necessary.

The *post_event(...)* function is used for inter-process communication. The function does not require any return values, the application can assume that the event is forwarded to the destined process. If the process is located on the local host, that process is called immediately with the event value. In all cases, the event is forwarded to the kernel node in case there are other nodes running the process to be notified. Once the kernel node receives such a notification, it sends one multicast message to deliver the event, thus reducing the delay.

The response times of the functions were as follows: 0.3 s, 0.19 s and 0.15 s for store, get and post operations, respectively. All the values are averages over all the hosts, over the multiple calls to the functions. Due to the cycle period being set to half second, it is not surprising that a store_data(...) is about 0.3 s (less than a cycle). A store operation takes longer, than a get function, since the later can be served also from the local cache. When the get_data(...) values were calculated, the local cache also improved the response times drastically. We will show later, how the cache influences the responsiveness of the system. The post_event function only requires one message to the kernel node, thus can reach the other nodes in the time-slot right after receiving it.

2) Power consumption: Along with the test application, we also used Contiki's powertrace [9] feature - this program logs the power consumption of the different hardware pieces on the device, such as the CPU (including the different power modes), memory access, sensor access and radio usage. Fig. 2 shows the current consumption of the host nodes, the kernel node, and the consumption of the traditional approach. The difference in current consumption is substantial: the kernel node needs to be powered up all the time, keeping the radio and the flash in use, while the host nodes are asleep, and only use their radios in their time-slots. The average power consumption of the kernel node was around 62000 uW, while that of the host node was a fraction of this, around 3500 uW - in other words, the host nodes use about 4% of the energy used by the kernel node. Looking at the average power consumption of the traditional implementation, it is about 4500 uW, almost a third more than ours. The host node initially has a high power consumption - this is due to the device keeping its radio on all the time until it synchronises its clock with the kernel node, and thus can start the TDMA-protocol. There is also a high peak at around 60 s - this is due to all the hosts changing from "store" mode to "retrieve", and notifying all the other nodes using the post notification, thus creating a high volume of traffic suddenly. It however calms down quickly. The two other, smaller peaks are also caused by transmission failures, and thus retransmissions. This is also visible on the dutycycle graph, shown next. The kernel node has a low initial power consumptionm because its radio is initially turned off while it initialises the storage system.
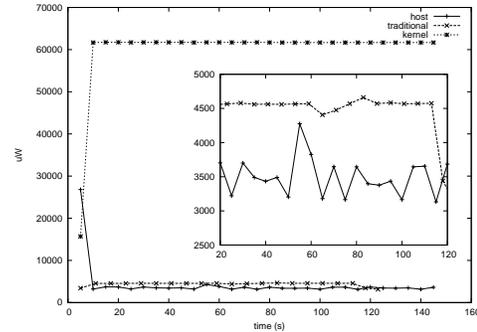

Fig. 2. Current consumption against time as logged by the powertrace application for a kernel, the host nodes and the traditional node during the test application. The innergraph shows a zoomed in picture of the traditional approach and the host node.

3) Radio dutycycle: We also logged the amount of time the radios were on, thus testing our communication protocol. Our aim was to spend as little energy as possible on the radio, since we know, with high probability, when we can expect transmissions. Our radio dutycycle was between 1% - 8% (with an average of 2.5%), which is considered very good in such a high-data rate application. Due to X-MAC, the dutycycle of the traditional implementation was around 2%, however the system failed when we used all the available nodes of the network due to interference and collisions - more than 60% of the messages had to be retransmitted. The reason is that X-MAC is a carrier-sense based protocol, i.e. it cannot guarantee timely delivery, but instead does a best-effort job, and it is very good at keeping the radio dutycycle low.

4) CPU usage: Another interesting way of looking at power consumption is the CPU usage. Ther kernel node used its CPU about 63% of the time, on average, while the traditional implementation had an average of 20% CPU usage. The host nodes' CPU usage was 1.5%. First of all, it is worth noting that the powertrace application itself uses some CPU time, however, as it was running on all the nodes, it influenced the nodes equally. Out of the three cases, the kernel node spent most time using its CPU, while the next one was the traditional implementation. It was a bit surprising to us, as we expected it to use the CPU, but the average was somewhat higher than expected. As it turned out, this was due to the X-MAC keeping the CPU busy - however, it may simply be due to the specific implementation of the protocol. The host nodes could spend the majority of time in low-power mode, thus resulting in a lower total power consumption.

From the previous two set of results, we can also concur the well-known phenomena: radio is the major energy consumer on the sensor nodes. A small improvement on the dutycycle can result in large savings when it comes to energy. Both the traditional and the host nodes consume a fraction of the energy consumed by the kernel node, since their radio has a very low dutycycle (it is only on about 2% of the time). It is also interesting that the host node uses its CPU much less
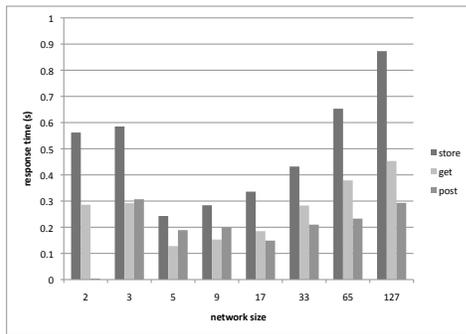
(a) Response times of different functions.
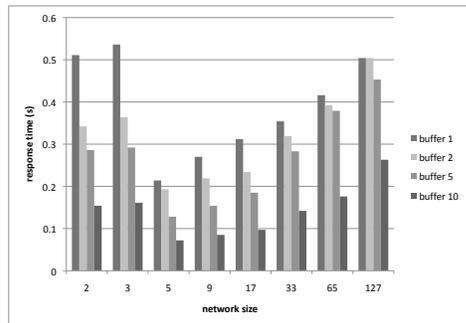
(b) Response time of the data_get(...) function for different buffer sizes.

Fig. 3.   Response times.

than the traditional approach (1.5 % against 20%), and this results in almost 50% energy savings.

D. Simulation-based evaluation

Although the testbed had relatively many nodes, considering our application scenario, we wanted to find out how far we can push the system in terms of scalability with the use of a networking simulator.

For the simulation, we use COOJA [22], Contiki's cross platform networking simulator. One of the features of COOJA is to import compiled binary images. In our case, we compiled our code for the Tmote platform, and imported the same code into COOJA, that we installed on the actual nodes.

1) Setup: We varied the number of nodes deployed randomly in a fixed-size area - the network size ranges from 2 to 128. The radio range was fixed so that all the nodes could see every other. We set the cycle period of the MAC protocol to half a second, and logged the time it takes for each function call to return to the application, as well as the energy consumption of each node. The cycle period is an important parameter since each host node has one slot in a cycle (though can possibly send multiple messages within one slot, depending on the length of the slot). When reliable communication is required between the host and the kernel, there are at least 3 message exchanges (request - reply - ack) - requiring at least one cycle period, i.e. the time period for the TDMA-protocol to assign the slot for the host again (see Section V).

2) Results: Fig. 3(a) shows the time it took for the three different functions to return to the caller for the different

network sizes. When there are very few nodes, the response time is dictated by the cycle period (0.5 s) - even if there are few nodes, they have to wait idle for the next cycle to reply to a message. On the other hand, the protocol works well for the number of nodes we expect to have (tens of nodes). When there are more than 100 nodes, the congestion becomes a problem, and the base station has a hard time keeping up with the requests within the short slot durations. When the expected number of nodes is higher than 100, the cycle period should be increased to give more time to the base station. Note, that even for 128 nodes, the response time is within 1 s, which is tolerable in our scenario.

In all cases, the store operation takes the longest. The charts show the time it takes to receive an acknowledgement from the store operation. The get function is quicker, as the local cache speeds up the response time. Post operation is generally the quickest, since the kernel node can notify the other nodes right after receiving the notification message. In some cases (for small network sizes), the get operation is quicker as a result of the cache being more influential to the response times as the delay due to network contention. For the 2 node case, the post gets delivered locally (i.e. the kernel does not have to notify any other node), therefore there is no delay.

Fig. 3(b) shows the response times of the *get_data(...)* function for different network sizes, and varying buffer sizes. The different buffer sizes show how many data chunks the node can hold locally. Intuitively, if we decrease the buffer size, the system will take longer to get the data, while increasing the buffer size results in shorter response times. In fact, this is what our simulations show. Therefore, the buffer size should be set based on the capabilities of the devices. In terms of energy usage, different buffer sizes do not make any difference, as the host nodes synchronise their buffers with the kernel node, regardless of its size - thus, the number of transmissions always equals to the amount of data being stored or retrieved.

VII. OPTIMISATIONS AND FUTURE WORK

The system implementation in this paper is our first attempt at tackling the problem, and thus contains some assumptions and simplifications. In this section, we would like to address these issues, present out future work on improving the system , and highlight the research topics that the other research groups can potentially pay attention to.

One of our current implementation limitation is that the kernel node has to be in reach of all the other host nodes. This was necessary to guarantee quick response times to system calls, and keep the hosts free of any extra work. If the system is to expand beyond the communication range of the kernel node, further kernel nodes would need to be placed to cover the area. We argue that such kernel nodes can be easily plugged into a wall socket (as they are envisaged to be desktop machines) or connect to a WiFi link (if they are mobile phones or laptops), and thus can communicate without much extra time overhead.

The other solution to the scalability problem is to have multi-hop communication between the hosts and the kernel.

We disregard this solution due to the extra complexity in the TDMA protocol, and due to the processing and memory overhead on the host nodes. In our current implementation, the host nodes use 4% of the energy used by the kernel nodes. If they have to be listening to other host nodes, this advantage is lost.

A different issue, that of reprogramming and application installation can also be further improved. In our current implementation, we install applications on nodes based in statically defined properties - i.e. the application needs to be tagged with information about its requirements: what kind of sensors it needs access to, etc. A smart compiler could recognise these requirements, and provide information for the kernel nodes automatically.

VIII. Conclusion

In this paper, we introduced OSone, our distributed operating system. The main contribution of the paper is the different way of looking at a sensor network: considering it one system helps designers produce more efficient and powerful applications, and do not have to spend the majority of their time optimising their application for the underlying operating system. We described our design of the architecture, as well as shown how it could be implemented on a well-known sensor platform for the 'smart-home' application scenario. The system was, then, quantitatively evaluated, and shown to be both energy and memory efficient, while staying responsive even as the network size grows.

We also gave room for future works as we believe the system has more potential, and can make a big difference to sensor networks on the field.

References

[1] "Imote2 Sensor board".
[2] "Tmote Sky Sensor board".
[3] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.
[4] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *SenSys '06: Proc. of the 4th Int. Conf. on Embedded Networked Sensor Systems*, pages 307–320, New York, NY, USA, 2006. ACM.
[5] M. Ceriotti, L. Mottola, G. P. Picco, A. L. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta, and P. Zanon. Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 277–288, Washington, DC, USA, 2009. IEEE Computer Society.
[6] J. Crowcroft. *Open Distributed Systems*. Artech House, Inc., Norwood, MA, USA, 1996.
[7] A. Dunkels. Rime — a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007.
[8] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, Nov. 2006.
[9] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Demo Abstract: Software-based Sensor Node Energy Estimation. In *Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems (SenSys 2007)*, Sydney, Australia, Nov. 2007.
[10] V. Dyo, S. A. Ellwood, D. W. Macdonald, A. Markham, C. Mascolo, B. Pásztor, S. Scellato, N. Trigoni, R. Wohlers, and K. Yousef. Evolution and sustainability of a wildlife monitoring sensor network. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 127–140, New York, NY, USA, 2010. ACM.
[11] J. V. Gruenen and J. Rabaey. Content management and replication in the snsp: A distributed service-based os for sensor networks. In *IEEE CCNC (Consumer Communications and Networking Conference)*, pages 655–659, January 2008.
[12] P. Hurni and et. al. A Testbed Management Architecture for Wireless Sensor Network Testbeds (TARWIS). In *Proceedings of the 7th European Conference on Wireless Sensor Networks (EWSN), Coimbra, Portugal*, 2009.
[13] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.
[14] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. Campbell. A survey of mobile phone sensing. *IEEE Communications Magazine*, 48(9):140–150, September 2010.
[15] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 149–162, New York, NY, USA, 2005. ACM.
[16] K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, and M. Welsh. Mercury: a wearable sensor network platform for high-fidelity motion analysis. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 183–196, New York, NY, USA, November 2009. ACM Press.
[17] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
[18] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.
[19] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-low Power Data Storage for Sensor Networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 374–381, New York, NY, USA, 2006. ACM.
[20] L. Mottola and G. P. Picco. Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks. In *Proc. of the Int. Conf. on Distr. Computing in Sensor Systems (DCOSS)*, 2006.
[21] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.
[22] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level Simulation in COOJA. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007.
[23] K. K. Rachuri, M. Musolesi, C. Mascolo, J. Rentfrow, C. Longworth, and A. Aucinas. EmotionSense: A Mobile Phones based Adaptive Platform for Experimental Social Psychology Research. In *Proceedings of 12th ACM International Conference on Ubiquitous Computing (UbiComp '10)*, Copenhagen, Denmark, September 2010. ACM.
[24] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, November 2008.
[25] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proc. of the Symp. on Networked Systems Design and Implementation (NSDI)*, 2004.