A SINGLE SYSTEM IMAGE JAVA OPERATING SYSTEM FOR SENSOR NETWORKS

Emin Gun Sirer Rimon Barr John C. Bicket Daniel S. Dantas
Computer Science Department
Cornell University
Ithaca, NY 14853
{egs, barr, bicket, ddantas}@cs.cornell.edu

Abstract

In this paper we describe the design and implementation of a distributed operating system for sensor networks. The goal of our system is to extend total system lifetime through power-aware adaptation for sensor networking applications. Our system achieves this goal by providing a single system image of a unified Java virtual machine to applications over an ad hoc collection of heterogeneous sensors. It automatically and transparently partitions applications into components and dynamically finds a placement of these components on nodes within the sensor network to reduce energy consumption and increase system longevity. This paper describes the design and implementation of our system and examines the question of *where* and *when* to migrate components in a sensor network. We evaluate two practical, power-aware, general-purpose algorithms for object placement, as well as an adaptive scheme for deciding the time granularity of object migration. We demonstrate that our algorithms can increase sensor network longevity by a factor of four to five by effectively distributing energy consumption and avoiding hotspots.

1. Introduction

Sensor networks simultaneously promise a radically new class of applications and pose significant challenges for application development. Recent advances in low-power, high-performance processors and medium to high-speed wireless networking have enabled large scale deployment of sensor nodes, capable of local processing and multi-hop communication. Many interesting sensing applications, however, entail collaboration between components distributed throughout an ad hoc network. For example, sensor networks are often composed of three types of components: sensors acting as data sources, information consumers operating as data sinks, and numerous intermediate filters for performing application-specific data processing. While the data sources and sinks may be coupled tightly to the nodes to which they are attached, there is often a high degree of freedom in the placement of the data processing components. This freedom, coupled with the dynamic environment posed by sensor networks, makes it difficult to find the optimal distribution of application components among nodes.

Adapting to dynamically changing conditions by changing the distribution of components across a network is critical for many distributed networking applications. For example, the resources available to components at each node, in particular the available power and bandwidth may change over time and necessitate the relocation of application components. Further, event sources that are being sensed in the external environment, such as tracked objects or chemical concentrations, may move rapidly, thereby shifting network loads and requiring applications to adapt by migrating components. Finally, an application's behavior might change, as in the transition from defensive to offensive mode in a battlefront sensing application, modifying its communication pattern and necessitating a reorganization of its deployed components within the network.

Currently, sensor networking applications either rely on a static assignment of components to nodes or use ad hoc, manual policies and mechanisms for migrating code and data in response to change. A static assignment of functionality to nodes simplifies application design by obviating code migration and reduces meta-traffic in the network by eliminating component mobility, but it also leads to non-adaptive, fragile and energy-inefficient systems. For example, many current-sensor networks perform all data aggregation and processing at a central node. Such a network will stall as soon as the critical nodes on the dataflow path run out of power or move out of transmission range. In contrast, manual approaches to code and data mobility suffer from being platform-dependent, error-prone and hard to develop. Each application using a manual approach needs to re-implement the same migration, monitoring and communication mechanisms, correctly, on every platform. Further, locally optimal policies pursued by applications that share a common network may lead to globally unstable and energy-inefficient behavior when they conflict. For instance, a high-priority submarine tracking application running on an acoustic sensor grid may be forced to behave sub-optimally by a lower-priority whale tracking application, if the applications use ad hoc mechanisms to adapt to the current power levels in the network. In essence, application-level adaptation suffers from building on an abstraction level that is too low. An operating system that provides the requisite mechanisms and policies for code mobility would not only simplify application development, but also ensure the integrity of system-wide goals in the face of multiple applications competing for resources.

Unlike distributed programming on the Internet, where energy is not a constraint, delay is low, and bandwidth is plentiful, physical limitations of sensor networks lead to some unique requirements. Technology trends indicate that the primary limitation of sensor networks is energy consumption, and communication is the primary energy consumer. Measurements from first-generation sensor nodes [Pottie & Kaiser 00] show that sending one bit may consume as much energy as executing 3000 instructions. This motivates a system that performs more computation to reduce communication.

In this paper, we outline the design of MagnetOS, a single system image (SSI) operating system for sensor networks. Based on our target domain of energy-constrained sensor networks, we identify the following set of design goals for an operating system.

- **Efficient:** The system should execute distributed sensor network applications in a manner that conserves power and extends system lifetime. Policies and mechanisms used for adaptation in the systems layer should not require excessive communication or power consumption.

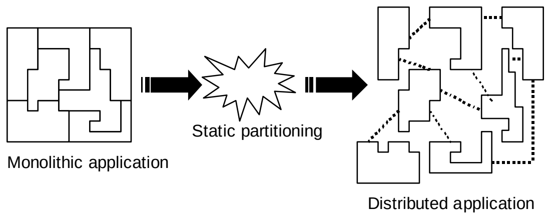

Figure 1: A static partitioning service converts monolithic Java applications into distributed applications that can run on an ad hoc network and transparently communicate via RPC.

- **Adaptive:** The system should respond automatically to significant changes in network topology, resource availability, and the communication pattern of the applications.
- **General purpose:** The system should support a wide range of applications and porting an existing centralized sensing application to execute efficiently on a sensor network should require little effort. Applications should be able to direct the adaptation using application-specific information. The system should provide effective default adaptation policies for applications that are not power-aware.
- **Platform independent:** Applications should be able to execute on ad hoc networks of heterogeneous nodes.

Our operating system meets these goals by providing the abstraction of a single unified Java virtual machine over an ad hoc network of heterogeneous, physically separate, potentially mobile sensor nodes. MagnetOS consists of a static application partitioning service that resides on border hosts capable of injecting new code into the network, and a runtime on each node that performs dynamic monitoring and component migration. The static partitioning service takes regular Java applications and converts them into distributed components that communicate via RPC [Birrell & Nelson 84] by rewriting them at the bytecode level (Figure 1). The code injector then finds a suitable initial layout of these components and starts the execution of the application. The runtime monitors the performance of the application and migrates application components when doing so would benefit the system.

The algorithms used to decide *where* and *when* to move application components form the core of our system. While MagnetOS is designed such that these algorithms can be transparently replaced to optimize for differing goals, such as minimizing application latency, response time, or bandwidth consumption, in this paper we tackle what we believe to be the most important goal in energy-constrained ad hoc networks of mobile hosts: we examine how to maximize total application and system lifetime by utilizing power more efficiently. We present two practical, online algorithms, named NetPull and NetCenter, for finding a distribution of application components on nodes in an ad hoc network that increases total system lifetime by increasing the effective energy utilization (Figure 2). We evaluate these algorithms in the context of a generic sensing application and examine their impact on system longevity, which we define as the length of time that a sensing application can maintain sensor coverage above a given threshold area. Both algorithms operate by dividing time into epochs, monitoring the communication pattern of the application components within each epoch, and migrating components at the end of the epoch when doing so would result in more efficient power utilization. We also present an online algorithm for selecting an epoch length that matches application behavior through online, adaptive statistical testing. We show that the MagnetOS system can achieve a factor of four to five improvement in system longevity over naive or static partitioning techniques.

This paper makes three contributions. It outlines the design and implementation of a single system image operating system for sensor networks, where the entire network appears to be a single Java virtual machine to applications, whose components are partitioned among the nodes automatically and migrated transparently at runtime. Second, we propose practical, adaptive, online algorithms for deciding *where* and *when* to move application components. Finally, we demonstrate that these algorithms achieve high-energy utilization, extract low overhead, and improve system longevity.

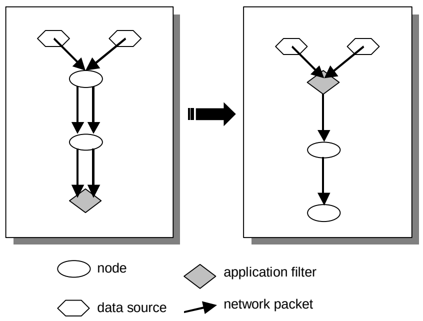

Figure 2: Migrating components closer to their data sources in a sensor network increases system longevity and decreases power consumption by reducing total network communication cost.

In the next section, we describe related work on operating system support for ad hoc sensor networks and their applications. Section 3 outlines our system implementation, including the code partitioning and distribution technique. Section 4 presents our network and application model, describes our simulation framework and evaluates NetPull and NetCenter within this environment. We summarize our contributions and results in Section 5.

2. Related Work

There has been much research on distributed operating systems, ad hoc sensor networks, and power management, though few systems have examined all three.

2.1. Distributed Systems

Single system image (SSI) operating systems have been examined extensively in the context of wired networks of workstations. Early landmark systems, such as V [Cheriton 88], Sprite [Ousterhout et al. 88], Ameoba [Tanenbaum et al. 90, Steketee et al. 95], Accent [Rashid & Robertson 81], and LOCUS [Popek & Walker 85], implemented native operating system facilities for migrating processes between nodes on a tightly coupled cluster. More recently, the cJVM [Aridor et al. 99] and JESSICA [Ma et al. 99] projects have examined how to extend a Java virtual machine-across a high-performance cluster. Others, including Condor [Litzkow et al. 97], libckpt [Plank et al. 95] and CoCheck [Stellner 96], provide user-level mechanisms for checkpointing and process migration without operating system support. These projects target high-performance, well-connected clusters. Their main goals are to balance load and achieve high performance in a local area network for interactive desktop programs or CPU-intensive batch jobs. In contrast, MagnetOS targets wireless multi-hop networks, where utilizing power effectively and maximizing system longevity is more important than traditional application performance.

Distributed object systems have examined how to support distributed computations in the wide area. Emerald [Jul et al. 88] provides transparent code migration for programs written in the Emerald language, where the migration is directed by source-level programmer annotations. Thor [Liskov et al. 93] provides persistent objects in a language-independent framework. It enables caching, replication and migration of objects stored at object repositories. More recently, grid computing systems such as Legion [Lewis & Grimshaw 95] and Globus [Foster & Kesselman 97] have focused on the construction of scalable, geographically distributed computing systems. These systems differ fundamentally from MagnetOS in that they require explicit programmer control to trigger migration, do not support an ad hoc network model and target traditional applications.

Some recent systems have focused on how to partition applications within a conventional wired network. The Coign system [Hunt & Scott 99] has examined how to partition COM applications between two tightly interconnected hosts within a local-area network. Coign performs static spatial partitioning of desktop applications via a two-way minimum cut based on summary application profiles collected on previous runs. Extending this work, the ABACUS system [Amiri et al. 00] has examined how to migrate functionality in a storage cluster. MagnetOS shares the same insight as Coign, in that it also focuses on the automatic relocation of application components, but differs in that it dynamically moves application components in response to changes in the network, instead of computing a static partitioning from a profile. [Kremer et al. 00] proposes using static analysis to select tasks that can be executed remotely to save energy. J-Orchestra [Tilevich & Smaragdakis 02] performs application partitioning via rewriting, leaving dynamic migration decisions under application control. Spectra [Flinn et al. 01] monitors resource consumption, collects resource usage histories and uses quality of service (fidelity) information supplied by the application to make resource allocation decisions. Spectra is invoked prior to operation startup, and statically determines a location at which to execute the operation.

Middleware projects have looked at constructing toolkits to support mobile applications. The Rover toolkit [Joseph et al. 95] provides relocation and messaging services to facilitate the construction of mobile applications. The Mobiware [Campbell 98] and DOMT [Kunz and Omar 00] toolkits are targeted specifically for ad hoc networks and provide an adaptive-QoS programming interface. XMIDDLE [Mascolo 01] assists with data management and synchronization. MagnetOS takes a systems approach instead of providing a programmer driven toolkit and automatically manages the shared network and energy resources among ad hoc sensor applications. This approach unifies the system layer and ensures that disparate applications, regardless of which toolkits they use, behave in a cooperative manner.

2.2. Ad hoc Routing Protocols

There has been much prior research on ad hoc routing algorithms. Proactive (e.g. DSDV [Perkins & Bhagwat 94], WRP [Murthy & Garcia-Luna_Aceves 96]), reactive (e.g. DSR [Broch et al. 98], AODV [Perkins 97], TORA [Park & Corson 98]) and hybrid (e.g. ZRP [Haas & MagnetOSman 98], HARP [Nikaein et al. 01]) routing protocols seek to pick efficient routes by proactively disseminating or reactively discovering route information, or both. While some protocols, such as PARO [Gomez et al. 01] and MBLR [Toh 01], have examined how to make power-aware routing decisions, all of these routing algorithms assume that the communication endpoints are fixed. Directed diffusion [Heidemann et al. 01] provides a data-centric programming model for sensor networks by labeling sensor data using attribute-value pairs and routing based on a gradient. MagnetOS complements the routing layer to move application code around the network, changing the location of the communication endpoints and radically altering the communication pattern of the overall application. It provides increased system and application longevity by bringing application components closer to the data sources, which complements the route selection performed by the ad hoc routing protocol.

2.3. Power Management

Prior work has also examined how to minimize power consumption within an independent host through various mechanisms [Pillai & Shin 01, Grunwald et al. 00, Weiser et al. 94, Douglis et al. 95, Stemm & Katz 96], including low-power processor modes, disk spin-down policies, adapting wireless transmission strength and selectively turning off unused devices. Our system is complementary to this work and opens up further opportunities for minimizing power consumption by shipping computation out of hosts limited in power to less critical nodes.

3. System Implementation and Distribution Model

MagnetOS implements a single system image operating system for sensor networks in two steps. First, a monolithic application is partitioned, distributing its functionality across the ad hoc network. The MagnetOS runtime then coordinates the communication and migration of these application segments across the nodes in the sensor network in order for the newly distributed application to appear as if running on a single Java virtual machine. We will now discuss the implementation details of the two components, the partitioning mechanism and the MagnetOS runtime.

3.1. Application Partitioning

The partitioning mechanism of MagnetOS converts Java applications written and compiled for a single virtual machine into remote objects that can be dispersed and executed across an ad hoc network of many virtual machines. The transformed application code, though modified to interact with the MagnetOS runtime, retains its original application semantics.

MagnetOS partitions applications at class granularity; consequently, the unit of mobility in MagnetOS is an object instance. This transformation at class boundaries preserves existing object interfaces. The entire transformation is performed at the byte-code level via binary rewriting, without requiring source-code access.

Our approach to partitioning applications statically is patterned after distributed virtual machines [Sirer et al. 99]. Static partitioning confers several advantages. First, the complex partitioning services need only be supported at code-injection points, and can be performed offline. Second, since the run-time operation of the system and its integrity do not depend on the partitioning technique, users can partition their applications into arbitrary components if they so choose. Further, since applications are verified prior to injection into the network, individual MagnetOS nodes need not re-run a costly verifier on application components. Finally, binary rewriting provides a convenient, default mechanism for transitioning legacy, monolithic applications to execute over ad hoc networks.

The static partitioning takes original application classes, and for each class, creates an instance (Magnet), a remote stub (MagnetPole), an interface (MagnetInterface) and a class object (MagnetStatic).

A Magnet is a modified implementation of the original class that stores the instance variables of the object. Each Magnet is an object instance and is free to move across nodes in the network. MagnetPoles, on the other hand, are remote references to the corresponding Magnet instance. That is, MagnetPoles are used to invoke procedure calls on remote Magnets residing on other nodes. Calls to the MagnetPole are intercepted by the MagnetOS runtime and converted into RPCs. This level of indirection enables code migration. As a Magnet moves, the method calls to the corresponding MagnetPoles are tracked by the MagnetOS runtime and directed to the new Magnet location. MagnetInterfaces capture the interface that the original class exposes to the rest of the application. Magnet and MagnetPole instances implement the MagnetInterface of the original class.

Several modifications to the application binaries are required for this remote object mechanism to work seamlessly. First, object creations (**`new`** instructions and matching constructor invocations) are replaced by calls to the local MagnetOS runtime. The runtime selects an appropriate node and constructs a new Magnet instance at that location. This operation returns a corresponding, properly initialized MagnetPole, which is then used in subsequent method invocations. In addition, MagnetOS adds accessor methods of the appropriate type for each field, converting field accesses into method calls to the MagnetPole reference, which subsequently remotely call the accessor on the corresponding Magnet instance. Finally, typechecking and synchronizing instructions (**`checkcast`**, **`instanceof`**, **`monitorenter`** and **`monitorexit`** instructions, and **`synchronized`** methods) are rewritten to trap into the MagnetOS runtime. The runtime reconstructs the appropriate type check on top of the modified type hierarchy, retaining the original application's behavior. Similarly, it converts lock acquisitions and releases into centralized operations at the Magnet. Unlike regular JVM monitor operations, each thread in MagnetOS is identified via a globally unique identifier, consisting of a <*nodeid, threadid*> tuple. This enables MagnetOS to identify cases of recursively locked mutexes and support the lock semantics required by the Java virtual machine.

The final component created for a class is a MagnetStatic object. It contains static field members, also known as class instances. In Java, static fields are shared across all instances of an object. The partitioning service coalesces the static fields of the original class into a single object, to which all the Magnet instances of that class retain a reference. Static field and method instructions are replaced with calls to the MagnetOS runtime, which tracks the location of the MagnetStatic object and forwards the static operations to the appropriate node in the ad hoc network.

While MagnetOS attempts to provide reasonable defaults for converting regular JVM applications to work in an ad hoc network, it explicitly does not try to make the process completely transparent and support network-oblivious applications. Applications running on networks have diverse failure modes that cannot be masked, and MagnetOS reflects such failures to the programmer by mapping them to implicit runtime exceptions, similar to the way other exceptional events, such as running out of memory, are handled. This mapping conforms to the JVM specification and provides a way for the application programmer to react to runtime events. We have found that most non-trivial, stateful applications will require some amount of failure recovery in all but the most densely connected and static networks.

3.2. Runtime Mechanisms for Object Migration

The MagnetOS runtime provides the dynamic services that facilitate the distributed execution of componentized applications across an ad hoc network. Its services include component creation, inter-component communication, object migration, garbage collection, naming, and object discovery. These runtime services are invoked in three ways: through background processes that automatically manage the running system, indirectly via MagnetPoles, and directly via explicit API calls from the application.

In order to create a new instance of an object, an application will contact the local runtime and pass the requisite type descriptor and parameters for object creation. The runtime then has the option of placing the newly created object at a suitable location with little cost. It may choose to locate the object on the local node, at a well-known node or at its best guess of an optimal location within the network. In our current implementation, all new objects are created locally. We chose this approach for its simplicity, and rely on our dynamic object migration algorithms to find the optimal placement of objects over time. Furthermore, short-lived, tightly scoped objects do not travel across the network unnecessarily. The application binaries, containing all of the object constructors, are distributed to all nodes at the time that the application is introduced into the network. Once created, the (remote) runtime simply initializes the object by calling its constructor and returns a MagnetPole instance referring to the (remote) object.

The runtime transparently handles invocations among the application components distributed across the network. Each runtime keeps a list of the live, local Magnets, which can optionally be named. MagnetPoles maintain the current location of their corresponding Magnet, and make runtime calls on behalf of application invocations to marshal arguments to, and results from, the appropriate node and object.

The MagnetOS runtime implements a lease-based garbage collector for remote objects, with leases automatically renewed by MagnetPoles. As in RMI and Network Objects [Birrell et al. 94], we do not collect cycles in the object reference graph. Local objects are handled by the standard Java garbage collector.

MagnetOS migrates application components at runtime by serializing Magnet state and moving it to a new node. MagnetPoles are informed of the relocation lazily, the next time they invoke a Magnet method or renew their object lease, via a forwarding reference left behind when an object migrates. Long chains of forwarding pointers, if allowed to persist for a long time, would pose a vulnerability – as nodes die, out-of-date MagnetPoles may not be able to trace a path to the current location of the object to which they are connected. MagnetOS collapses these paths whenever they are traversed. Periodic lease updates in lease-based garbage collection requires periodic communication between MagnetPoles and Magnets, which provides an upper-bound on the amount of time such linear chains are permitted to form in the network.

The MagnetOS runtime provides an explicit interface by which application writers can manually direct component placement. This interface allows programmers to establish affinities between components and ad-hoc nodes. We provide two levels of affinity. Specifying a “strong” affinity between a component and a node effectively anchors the code to that node. This is intended for attaching components like device drivers to the nodes with the installed device in them. Specifying a “weak” affinity immediately migrates the component to the named node, and allows the automated code placement techniques described in the next section to adapt to the application’s communication pattern from the new starting point. Note that today’s manually constructed applications correspond to the use of strong affinity in our system – unless explicitly moved, components are bound to nodes. The result of overusing strong affinity is a fragile system, where unforeseen communication and mobility patterns can leave an application stranded. While we provide these primitives, we do not advocate their use and believe that automated techniques can outperform manual efforts to place components.

3.3. Runtime Support for Ad hoc Networks

The sensor networking domain places additional constraints on the runtime implementation.

First, multi-hop ad hoc networks require an ad hoc routing protocol to connect non-neighboring nodes. MagnetOS relies on a standard ad hoc routing protocol below the runtime to provide message routing. Currently, our system runs on any platform that supports Java JDK1.4. On Linux, we use an efficient in-kernel AODV implementation we developed. On other platforms, we use a user-level version of AODV written in Java to provide unicast routing. The choice of a routing algorithm is independent from the rest of the runtime, as the runtime makes no assumptions of the routing layer besides unicast routing.

In addition, standard communication packages such as Sun’s RMI are designed for infrastructure networks, and are inadequate when operating on multi-hop ad hoc networks. Frequent changes in network topology and variance in available bandwidth require MagnetOS to migrate objects. However, standard RMI does not provide an easy-to-use mechanism by which the endpoints of an active connection can be modified. Consequently, we have had to develop our own RPC package. Similar to Sun’s RMI, it supports a synchronous interface and uses a reliable datagram protocol resembling RDP [Hinden & Partridge 90] instead of TCP. The MagnetOS RPC package allows us to easily modify the communication endpoints when components move and is responsible for all communication between MagnetPoles and corresponding Magnets.

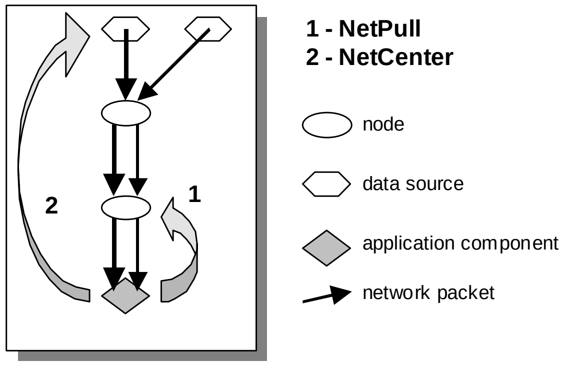

Figure 3: NetPull moves one hop towards the source of data whereas NetCenter moves directly to the source of most packets.

Finally, the higher-level policies in MagnetOS require information on component behavior to make intelligent migration decisions. The runtime assists in this task by collecting, for each component, information on the amount of data it exchanges with other components. The runtime intercepts all RPCs and records, for all incoming *and* outgoing invocations per component, the source and destination. It keeps a cumulative sum per component per epoch, and periodically informs the migration policy in the system of the current tally. While this approach has worst case space requirement that is $O(N^2)$, where N is the number of components in the network, most components communicate with few others, and the space requirements are typically small. For instance, in the sensor benchmark examined in Section 4, the storage requirements are linear. The next section describes how MagnetOS uses these statistics to automatically migrate components.

3.4. Finding an Energy-Minimizing Object Placement

In this section, we describe two algorithms, NetPull and NetCenter, which use the information gathered by the runtime to migrate components in a manner that increases system longevity.

Both NetPull and NetCenter share the same basic insight. They shorten the mean path length of data packets by automatically moving communicating objects closer together. They perform this by profiling the communication pattern of each application in discrete time units, called epochs. In each epoch, every runtime keeps track of the number of incoming and outgoing packets for every object.

At the end of each epoch, the migration algorithm decides whether to move that object, based on its recent pattern of behavior. Under both algorithms, the decision is made locally, based on information collected during recent epochs at that node. NetPull and NetCenter differ in the type of information they collect and how they pick the destination host. Depending on the environment, one may be easier to implement.

NetPull collects information about the communication pattern of the application at the physical link level, and migrates components over physical links one hop at a time. This requires very little support from the network; namely, the runtime needs to be able to examine the link level packet headers to determine the last or next hop for incoming and outgoing packets, respectively. For every object, we keep a count of the messages sent to and from each neighboring node. At the end of an epoch, the runtime examines all of these links and the object is moved one hop along the link with greatest communication.

NetCenter operates at the network level, and migrates components multiple hops at a time. In each epoch, NetCenter examines the network source addresses of all incoming messages, and the destination addresses of outgoing messages for each object. This information is part of the transmitted packet, and requires no additional burden on the network. At the end of an epoch, NetCenter finds the host with which a given object communicates the most and migrates the object directly to that host.

Both of these algorithms improve system longevity by using the available power within the network more effectively. By migrating communicating components closer to each other, they reduce the total distance packets travel, and thereby reduce the overall power consumption. Further, moving application components from node to node helps avoid hot spots and balance out the communication load in the network. As a result, both algorithms can significantly improve the total system longevity for an energy-constrained ad hoc network.

3.5. Determining Adaptation Granularity

Both NetPull and NetCenter are epoch-based algorithms, and are therefore sensitive to the time granularity at which they adapt. Adapting too quickly can result in wasted energy due to poor migration decisions in response to transient conditions and normal system perturbation. Adapting too slowly can also result in wasted energy when significant changes in application behavior or the environment are not addressed.

The insight guiding our approach to epoch selection is that a well-placed component should get equal numbers of packets from all directions within an epoch. Consequently, a distinct direction that dominates the communication pattern marks the boundary of an epoch. We implement an adaptive epoch selection algorithm based on a statistical test that determines whether the data collected thus far is sufficient to make a migration decision with high confidence. We accumulate the number of packets sent to each component by its communicating peers, and then compute the likelihood of this event occurring from a multinomial distribution with equally weighted priors as follows:

$$P(x_1,\ldots,x_n) = \left(\sum x_i\right)! \cdot \frac{\prod (p_i)^{x_i}}{\prod (x_i)!}$$

where: n – number of peers
x_i – number of packets from node i
p_i – $1/n$.

If the probability calculated above is below a threshold value, we continue to accumulate data, and defer the migration decision. We migrate components only when we can determine with high probability that one direction in the network dominates the communication pattern.

4. Evaluation

In this section, we evaluate the performance of MagnetOS. We first evaluate the core object migration algorithms, NetPull and NetCenter, and show that they achieve good energy utilization, improve system longevity, and are thus suitable for use in a general-purpose, automatic object migration system. We then discuss the benefits of epoch length adaptation. Finally, we report results from some microbenchmarks to show that automatically partitioning applications does not extract a large performance cost, and that the memory costs of a specially-tuned Java virtual machine is within the resource-budget of next generation sensors.

4.1. Simulation Framework

Our system targets general-purpose sensor networks. We developed a fast, packet-level, statically parameterizable ad hoc networking simulator in order to simulate large networks. The simulator accounts for all communication costs, including AODV routing and route repair overhead. It models the movement of every unicast and multicast packet and incurs the cost of moving a condenser and notifying all its sensors of the new location.

We initialize the simulator with a uniform distribution of nodes on a plane, and vary parameters such as noise levels, field size, density, battery power, and communication and sensing radii. Sensing events are generated at random locations on the field, and with random durations and velocity vectors. Sensors that are in range detect these signals and generate application events. They can also, with small probability, generate fictitious events due to sensor noise.

4.2. Network Model and Benchmark Application

In our simulations, all nodes have the same communication radius and they are connected to the fixed networking infrastructure via a single, centrally placed node. Each node initially stores a fixed, finite amount of energy. Sending a packet between any neighboring nodes exacts a constant communication cost, and the cost of local computation on a host is negligible in comparison.

We examine a generic, reconfigurable sensing benchmark we developed named SenseNet. This application consists of sensors, condensers and displays. Sensors are fixed at particular ad hoc nodes, where they monitor events within their sensing radius and send a packet to a condenser in response to an event. Condensers can reside on any node, where they process and aggregate sensor events and filter noise. The display runs on the central node, extracts high-level data out of the sensor network and sends it to the wired network.

4.3. Algorithms

We compare four different algorithms for automatic object migration:

- *Static* corresponds to a static, fixed assignment of objects to nodes within the network. Our components remain at the home node for the entire duration of the simulation.
- *Random* selects a random destination for each component at each epoch. It corresponds to a simple load-balancing algorithm, designed to avoid network hotspots.
- *NetPull* moves components one hop along the most active adjacent communication link at each epoch to the most active neighbor.
- *NetCenter* moves components directly to the node with greatest activity in the previous epoch.

4.4. Simulation Parameters

A simulation of a complex system such as this requires many parameters. In the following experiments, we model a terrestrial sensor network consisting of nodes with seismic sensors for object tracking. We examine large sensor networks, consisting of 3600 nodes. The field size is 300 by 300 distance units, which, if scaled to a meter, correspond to a density of 0.04 sensors/m^2, a practical density for distribution from the air. Node sensing radius is 20 units; communication radius is 10. Sensors generate spurious readings, or noise, due to local vibrations and sensor noise; we assume that sensors are fairly accurate, and that only 1% of the messages are attributable to noise. We have examined, but do not present results from, simulations with higher sensor density (0.02 through 0.06) and noise level (1% through 10%). The choice of a particular density, or noise level does not qualitatively impact our results; they shift the curves without affecting any of the trends. The choice of epoch duration is initially fixed – we address the epoch length selection question separately in Section 4.6. Each epoch contains at least one event in our simulations, where an event corresponds to an object being tracked, and moves through the sensor field at a randomly chosen velocity. An event may span up to 10 epochs and move through the field at a velocity between 0.0 and 2.0 distance units per epoch. We assume that nodes are stationary after the initial deployment, though none of the nodes make any assumptions about geographical location of other nodes. Every data point represents an average of five runs.

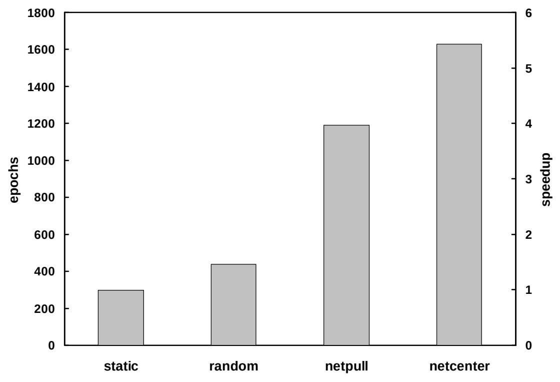

Figure 4: System longevity improvement.

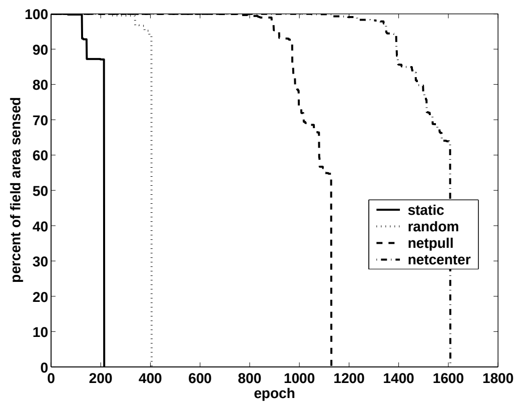

Figure 5: Sensor coverage degradation over time.

4.5. Results and Discussion

Figure 4 illustrates the impact of our algorithms on system longevity. In this simulation, we define system failure as the point when half of the field is no longer being sensed, that is, only half of the field area is within the sensing radius of at least one live sensor which can communicate along some functioning route with the home node. Static corresponds to current, naïve implementations of sensing applications, where all data is pooled off of the sensor network for processing on a central node. The network becomes unoperational as soon as the gateway nodes around the central node run out of power. Random performs better by 50% because it distributes the load more evenly and avoids hot spots. NetPull and NetCenter lengthen the operational lifetime of the system by a factor of four to five by performing in-network processing at suitably-selected locations.

Figure 5 shows how quickly the field coverage degrades when components are assigned to nodes in a manner that is oblivious to the underlying application communication pattern. In contrast, NetPull and NetCenter migrate components close to the source of events, thus preserving energy and shedding load from the critical nodes in the network.

The slopes of the curves in Figure 6 demonstrate that actively migrating components in the network drains fewer nodes than a static placement. This is because active migration algorithms avoid creating hotspots around critical nodes. In addition, both Static and Random reach system failure with far fewer drained nodes, indicating that these algorithms distribute load unevenly and lead to hotspots.

Figure 7 shows that NetPull and NetCenter lie between Static and Random in terms of energy consumption per epoch. The low slope for Static shows that this algorithm uses less energy per epoch because it does not expend any power on ac-

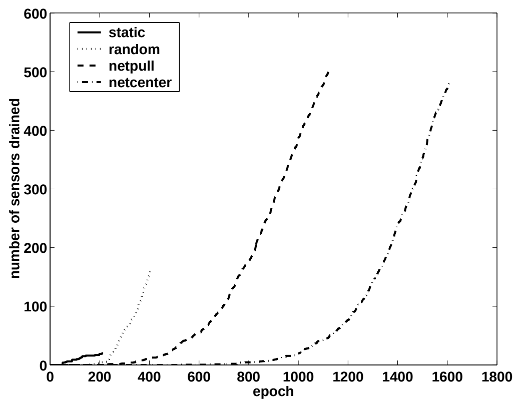

Figure 6: Sensor drainage over time.

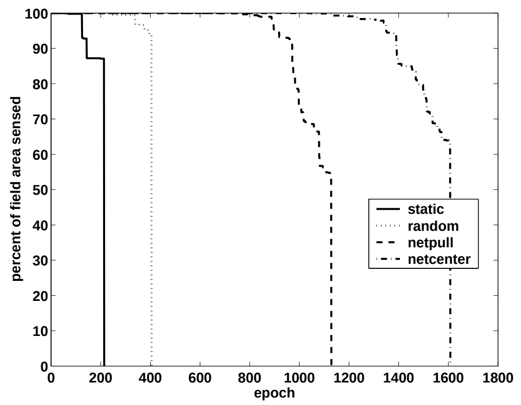

Figure 7: Field energy used over time.

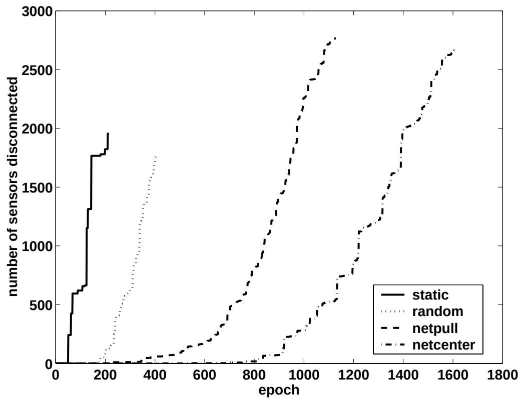

Figure 8: Sensors disconnected over time.

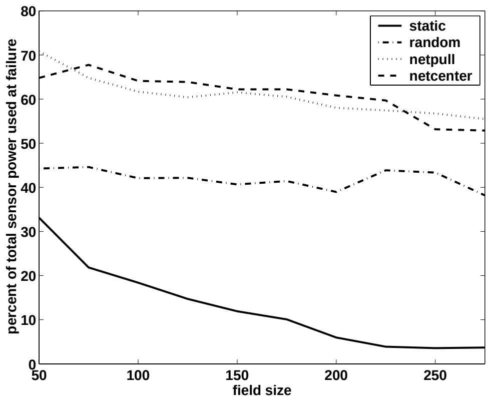

Figure 9: Energy used at breakdown vs. field size.

tive object migration. Most of this power usage is concentrated in a ring around critical nodes, however, and the application terminates early, leaving more than 95% of the total energy on the field unutilized. NetPull and NetCenter consume more energy than Static, because they need to move components, but consume less energy than Random, because they take application behavior into account. Overall, they outlast both random and static despite lying between them.

The graph of disconnected nodes over time, shown in Figure 8, indicates that the number of disconnected nodes increases more gradually for NetPull and NetCenter because they distribute load more evenly across the network. In the case of Static and Random, even though only a small number of nodes are drained, they are all located around the home node and thus quickly disconnect the entire field from the wired network.

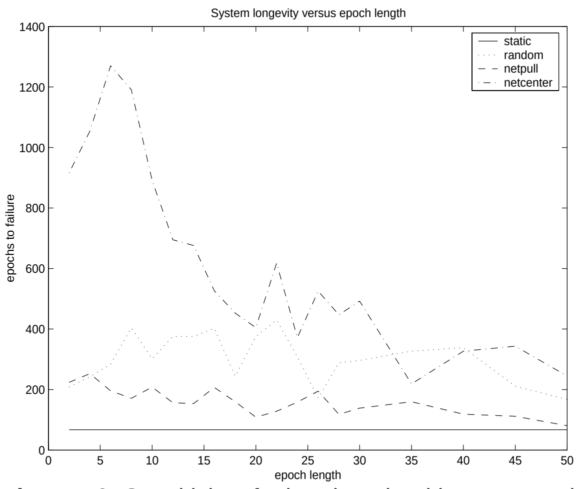

Figure 10: Sensitivity of migration algorithms to epoch length.

Finally, Figure 9 shows that all algorithms, except for the static placement of components, are unaffected by variations of field size, and will scale to large networks. Static does not scale, because the number of critical nodes is a constant function of the application data flow graph, and is not proportional to the size of the field.

Overall, NetPull and NetCenter achieve good energy utilization and improved system longevity. Their simplicity makes them strong candidates for use in automated object migration systems.

4.6. Epoch length selection

In this section, we evaluate epoch length selection algorithm described in section 3.5. To test the sensitivity of object migration to epoch length, we set up an experiment where the field contains four event sources located at the midpoint of each edge. Events are generated for half a 10 second period in a pair-wise alternating manner. Figure 10 evaluates the performance of automatic object migration algorithms at different, statically fixed epoch lengths. It demonstrates the dangers of an epoch length that is not matched to the temporal communication patterns in the network. A highly adaptive algorithm like NetCenter performs best when its epoch length matches that of the underlying event source. At shorter epoch lengths it wastes energy by performing excessive migrations. At harmonics of the beaconing frequency, some of the migration algorithms perform slightly better, though at long epoch lengths all the migration algorithms fail to adapt to network changes.

In contrast with statically selected epoch lengths, yhe adaptive epoch length selection algorithm described in section 3.5 achieves an average system longevity of 2370 when combined with NetCenter.

It outperforms any static epoch length selection shown in Figure 10 by 50%.

4.7. MagnetOS RPC performance

An automatic approach to application partitioning and transparent object migration would be untenable if the performance of automatically partitioned applications suffered significantly. In this section, we show that the performance of automatically partitioned and rewritten applications is competitive with manually partitioned applications. In the micro-benchmark below, we compare the overhead of our RPC implementation to that of remote invocations performed via Java RMI, on a 1.7 GHz P4 with 256 MB of RAM JDK 1.4 on Linux 2.4.17 with AODV. On all micro-benchmarks, automatically decomposed applications are competitive with manually coded, equivalent RMI implementations.

Remote call	Java RMI	MagnetOS
Null	430 ± 16	172 ± 6
Int	446 ± 9	180 ± 8
Obj. w/ 32ints	991 ± 35	174 ± 4
Obj. w/ 4int, 2obj	844 ± 21	177 ± 7

all times in µs, average of 1000 calls.

Table 1: Remote method invocation comparison.

4.8. Space-optimized Java Virtual Machine

The applicability of a Java-based SSI OS is limited by the ability of sensor network nodes to support the requisite services of a Java VM. Java virtual machines on the desktop can indeed have excessive resource requirements; the Java virtual machine we started out with, the Kimera VM, also had significant memory requirements. Table 2 shows the size of different Java virtual machines at application startup.

Virtual Machines	Peak Memory Consumption
Sun Java JDK 1.4	9000 KB
Kimera Unmodified	22822 KB
Kimera Optimized	1172 KB
Sun J2ME KVM	160 KB
Sun Java Card VM	512B + 16 KB ROM

Table 2: Space consumption of Java virtual machines.

Traditional virtual machines, such as Sun Java JDK 1.4 and the original Kimera VM support many features and are not optimized for space. Consequently, they have high memory requirements and are not suitable for sensor networks. However, through simple space optimizations, including lazy loading, discarding basic block information, stack reduction, and eliminating the space allocated for reserved but unused fields, we have reduced these resource requirements 20-fold without compromising any VM functionality.

Even further space savings can be achieved by trading off functionality for space. For instance, Sun's J2ME KVM and Java Card VM have been specifically targeted for embedded platforms. They consume less memory by supporting fewer Java libraries, not providing features like code verification and reflection, and modifying the programming model to avoid rich data types. For instance, the Java Card VM can execute Java applications using only 512 bytes. Overall, we consider the problem of running a Java interpreter, or equivalent functionality, on sensors to be solvable. Our VM is capable of executing the MagnetOS runtime and Java applications with full access to the complete Java class libraries with 2 MB of RAM, which we expect to be available on sensor nodes of the near future.

5. Conclusion

In this paper, we present the design and implementation of a single system image operating system for sensor networks. Our system implements the Java Virtual Machine interface on top of a collection of sensor nodes. An application partitioning tool takes monolithic Java applications and converts them into distributed, componentized applications. A small runtime on each node is responsible for object creation, invocation and migration. We rely on a transparent RPC for node-independent communication between components. Overall, this distributed system provides a well-understood programming model for sensor network applications, while simultaneously providing the system with sufficient freedom to transparently move components in order to extend achieve power savings and extend sensor network lifetime.

We propose algorithms for automatically determining where to locate application components in the network to minimize energy consumption and to determine when to migrate application components by adaptively selecting an epoch duration. Combined, these algorithms enable MagnetOS to find an assignment of components to nodes that yields good utilization of available energy in the network. These algorithms are practical, entail low overhead and are easy to implement because they rely only on local information that is readily available. We have demonstrated that they can conserve power and achieve a factor of four to five improvement in system longevity.

Ad hoc sensor networking is a rapidly emerging area with few established mechanisms, policies and benchmarks. We hope that high-level abstractions, such as single system image operating systems combined with automatic object migration algorithms, will create a familiar and power-efficient programming environment, thereby enabling rapid development of platform-independent, power-adaptive applications for sensor networks.

References

[Amiri et al. 00] Khalil Amiri, David Petrou, Greg Ganager and Garth Gibson. Dynamic Function Placement in Active Storage Clusters. USENIX Annual Technical Conference, San Diego, CA, June 2000.

[Aridor et al. 99] Yariv Aridor, Michael Factor and Avi Teperman. cJVM: a Single System Image of a JVM on a Cluster. IEEE International Conference on Parallel Processing, September 1999.

[Birrell & Nelson 84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. ACM Transactions on Computer Systems, 2(1):39--59, February 1984.

[Birrell et al. 94] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. SRC Tech Report 115, Feb 1994.

[Broch et al. 98] J. Broch, D. B. Johnson, and D. A. Maltz, The Dynamic Source Routing Protocol for Mobile Ad hoc Networks. Internet-Draft, draft-ietf-manet-dsr-01.txt, Dec. 1998.

[Cheriton 88] David Cheriton. The V Distributed System. Communications of the ACM, 31(3), March 1988, pp.314-333.

[Douglis et al. 95] Fred Douglis, P. Krishnan and Brian Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In 2nd USENIX Symposium on Mobile and Location-Independent Computing, April 1995.

[Flinn 01] Jason Flinn, Dushyanth, Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany, May 2001.

[Foster & Kesselman 97] I. Foster, C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. Intl J. Supercomputer Applications, 11(2):115-128, 1997.

[Gomez et al. 01] J. Gomez, A. T. Campbell, M. Naghshineh and C. Bisdikian. PARO: Conserving Transmission Power in Wireless Ad hoc Networks. In Proceedings of the 9th International Conference on Network Protocols, Riverside, California, November 2001.

[Grunwald et al. 00] Dirk Grunwald, Philip Levis, Keith I. Farkas, Charles B. Morrey III and Michael Neufeld. Policies for Dynamic Clock Scheduling. In Proceedings of the Fourth OSDI, San Diego, California, October 2000.

[Haas & MagnetOSman 98] Z. J. Haas and M. R. MagnetOSman, The Zone Routing Protocol (ZRP) for Ad hoc networks (Internet-Draft). Mobile Ad hoc Network (MANET) Working Group, IETF, Aug. 1998.

[Harold 00] Harold, E. R. Java Network Programming. O'Reilly & Associates, Aug 2000.

[Hinden & Partridge 90] B. Hinden and C. Partridge, "Version 2 of the reliable data protocol (RDP)," RFC 1151, IETF, Apr 1990.

[Hunt & Scott 99] Galen C. Hunt and Michael L. Scott. The Coign Automatic Distributed Partitioning System. In Proceedings of the Third Symposium on Operating System Design and Implementation, pp. 187-200. New Orleans, Louisiana, February 1999.

[Joseph et al. 95] Anthony D. Joseph, Alan F. De Lespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek., Rover: A Toolkit for Mobile Information Access. In Proceedings of the Fifteenth SOSP, Dec 1995.

[Jul et al. 88] Eric Jul, Henry Levy, Norman Hutchinson, Andrew Black. Fine-Grained Mobility in the Emerald System. ACM TOCS, 6(1), Feb. 1988, pp. 109-133.

[Kremer et al. 00] U. Kremer, J. Hicks, and J. Rehg. Compiler-directed remote task execution for power management: A case study. Workshop on Compilers and Operating Systems for Low Power, PA, October 2000.

[Kunz & Omar 00] T. Kunz and S. Omar. A Mobile Code Toolkit for Adaptive Mobile Applications. IEEE Workshop on Mobile Comp. Syst. and Apps, Monterey, CA Dec 2000.

[Lewis & Grimshaw 95] Mike Lewis and Andrew Grimshaw. The Core Legion Object Model. Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, August 1995.

[Liskov et al. 92] Barbara Liskov and Mark Day and Liuba Shrira. Distributed Object Management in Thor. In Proc. of the International Workshop on Distributed Object Management, 1992, pp. 79-91.

[Litzkow et al. 97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report #1346, University of Wisconsin-Madison, April 1997.

[Lorch & Smith 98] Jacob R. Lorch and Alan Jay Smith. Software Strategies for Portable Computer Energy Management. IEEE Personal Communications Magazine, 5(3), June 1998.

[Ma et al. 99] Matchy J. M. Ma, Cho-Li Wang, Francis C. M. Lau and Zhiwei Xu. JESSICA: Java-Enabled Single System Image Computing Architecture. The International Conference on Parallel and Distributed Processing Techniques and Applications, June 1999.

[Mascolo 01] Cecilia Mascolo, Licia Capra and Wolfgang Emmerich. XMIDDLE - A Middleware of Ad hoc Networks. UCL-CS Research Note 00/54, 2001.

[Murthy & Garcia-Luna-Aceves 96] S. Murthy and J.J. Garcia-Luna-Aceves. An Efficient Routing Protocol for Wireless Networks. ACM Mobile Networks and App. J., Special Issue on Routing in Mobile Communication Networks, Oct. 1996, pages 183-97.

[Nikaein et al. 01] Navid Nikaein, Christian Bonnet and Neda Nikaein. HARP - Hybrid Ad hoc Routing Protocol. In Proc. of the International Symposium on Telecommunications, 2001.

[Ousterhout et al. 88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. IEEE Computer, 21(2):23--36, February 1988.

[Park & Corson 98] Vincent D. Park and M. Scott Corson. Temporally-Ordered Routing Algorithm (TORA) version 1: Functional Specification. Internet-Draft, draft-ietf-manet-tora-spec01. txt, August 1998.

[Perkins 97] Perkins, C.E. Ad hoc On-Demand Distance Vector (AODV) Routing. IETF MANET, Internet Draft, Dec.1997.

[Perkins & Bhagwat 94] C. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. ACM SIGCOMM, October 1994.

[Pillai & Shin 01] Padmanabhan Pillai and Kang G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. SOSP 2001, pp. 89-102.

[Plank et al. 95] James S. Plank, Micah Beck, Gerry Kingsley and Kai Li. Libckpt: Transparent Checkpointing under Unix. Usenix Winter 1995 Technical Conference, New Orleans, LA, January 1995.

[Popek & Walker 85] G. Popek and B. Walker, eds. The LOCUS Distributed System Architecture. MIT Press, Cambridge, MA 1985.

[Pottie & Kaiser 00] G.J. Pottie and W.J. Kaiser. Wireless integrated network sensors. Communications of the ACM, 43(5):51--58, May 2000.

[Rashid & Robertson 81] Rashid, R.F., Robertson, G.G. Accent: A Communication Oriented Network Operating System Kernel. 8th ACM SOSP. Pacific Grove, California, 1981.

[Sirer et al. 99] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory and Brian N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. 17th SOSP, South Carolina, December 1999.

[Steketee et al. 95] Chris Steketee, Piotr Socko, Bartosz Kiepuszewski. Experiences with the Implementation of a Process Migration Mechanism for Amoeba. In Proceedings of the 19th Australasian Computer Science Conference, January 1995, pp. 213-224.

[Stellner 96] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. International Parallel Processing Symposium, pp. 526--531, Honolulu, HI, April 1996.

[Stemm & Katz 96] Mark Stemm and Randy Katz. Measuring and Reducing energy consumption of network interfaces in hand-held devices. 3rd International Workshop on Mobile Multimedia Communications, Sept. 1996.

[Tanenbaum et al. 90] Tanenbaum, A.S., Renesse, R. van, Staveren, H. van., Sharp, G.J., Mullender, S.J., Jansen, A.J., and Rossum, G. van: Experiences with the Amoeba Distributed Operating System, Commun. ACM, vol. 33, pp. 46-63, Dec. 1990.

[Tennenhouse & Wetherall 96] D. L. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. In Multimedia Computing and Networking, San Jose, California, January 1996.

[Tilevich & Smaragdakis 02] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. European Conference on Object-Oriented Programming, 2002.

[Toh 01] C.K. Toh. Maximum Battery Life Routing to Support Ubiquitous Mobile Computing in Wireless Ad hoc Networks. IEEE Communications, June 2001.

[Weiser et al. 94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In Proc. of the First OSDI, Monterey, California, November 1994.