DESIGN AND IMPLEMENTATION OF A SINGLE SYSTEM IMAGE OPERATING SYSTEM FOR AD HOC NETWORKS

Hongzhou Liu Tom Roeder Kevin Walsh Rimon Barr Emin Gün Sirer
Department of Computer Science, Cornell University
liuhz,tmroeder,kwalsh,barr,egs@cs.cornell.edu

ABSTRACT

In this paper, we describe the design and implementation of a distributed operating system for ad hoc networks. Our system simplifies the programming of ad hoc networks and extends total system lifetime by making the entire network appear as a single virtual machine. It automatically and transparently partitions applications into components and dynamically finds them a placement on nodes within the network to reduce energy consumption and to increase system longevity. This paper describes our programming model, outlines the design and implementation of our system and examines the energy efficiency of our approach through extensive simulations as well as validation of a deployment on a physical testbed. We evaluate practical, power-aware, general-purpose algorithms for component placement and migration, and demonstrate that they can significantly increase system longevity by effectively distributing energy consumption and avoiding hotspots.

1 Introduction

Ad hoc 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 new applications for ad hoc and sensor networks, ranging from large-scale environmental data collection to coordinated battlefront and disaster-relief operations. Ad hoc networking applications differ from traditional applications in three fundamental ways. First, ad hoc networking applications are inherently distributed. Operating on a distributed platform requires mechanisms for remote communication, naming, and migration. Second, ad hoc networks are typically highly dynamic and resource-limited. Key performance metrics, such as bandwidth, may vary through several orders of magnitude, and mobile nodes are typically limited in energy. Consequently, applications need policies for using available resources efficiently, and sharing them among competing applications fairly. Finally, ad hoc networking applications are expected to outlast the lifetime of any one node. Performing long-running computations in a dynamic environment requires facilities for dynamically introducing new functionality and integrating it with existing computations present in the network. Current operating systems, however, provide little support for ad hoc networks.

The current state of the art in developing applications for ad hoc networks is to treat the network as a system of standalone systems, that is, a network comprised of independent, autonomous computers. This programming model forces applications to provide all of their requisite mechanisms and policies for their operation themselves. Mechanisms, such as those for distributing code and migrating state, as well as policies, such as how to react to diminishing battery supply on a given node, need to then be embedded, independently, in all applications. Such a limited programming model not only makes developing ad hoc networking applications tedious and error-prone, but the lack of a global operating system acting as a trusted arbiter between mutually distrusting applications allows inter-application conflicts to emerge. Critical global properties of the network, such as system longevity, are dictated by distributed policies encoded in applications; network operators have little control over the operation of their systems, as there is no network-wide system layer. This situation is analogous to the early standalone operating systems implemented entirely in user-level libraries, in that assuring global properties of the system requires whole system analysis, including auditing all application code.

In this paper, we investigate an alternative programming model for ad hoc networks where a thin distributed operating system layer makes the entire network appear to applications as a single virtual machine. We describe the design and implementation of a distributed operating system based on this model, called MagnetOS. We show that the high level of abstraction presented to applications with such a model not only simplifies the development of applications substantially, but also enables the underlying system to make energy-efficient placement and migration decisions. Unlike distributed programming on the Internet, where energy is not a constraint, delay is low, and bandwidth is plentiful, physical limitations of ad hoc networks lead to some unique requirements. Technology trends indicate that the primary limitation of mobile ad hoc and sensor networks is energy consumption, and communication is the primary energy consumer [38]. Consequently, the goals of MagnetOS are as follows:

Efficiency: The system should execute distributed ad hoc 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.

Adaptation: The system should respond automatically to significant changes in network topology, resource availability, and the communication pattern of the applications. Adaptation should not require a priori involvement from the application programmer.

Generality: The system should support a wide range of applications. Developing new applications that execute efficiently on an ad hoc network should require little effort. The system should provide effective default adaptation policies for applications that are not power-aware. Applications should be able to direct and, when executed with sufficient privilege, override the default adaptation using application-specific information.

Extensibility: The system should provide facilities for deploying, managing and modifying executing applications whose lifetime may exceed those of the network participants.

Compatibility: The system should not require mastering a new paradigm in order to deploy applications. Standard development tools should continue to work in building applications for ad hoc networks. The system should enable applications to execute on ad hoc networks of heterogeneous nodes.

MagnetOS meets these goals by making the entire network operate as an extended Java virtual machine. MagnetOS applications are structured as a set of interconnected, mobile event handlers, specified statically by the programmer as objects in an object-oriented system. The MagnetOS runtime uses application partitioning and dynamic migration to distribute the event handlers to nodes in the ad hoc network and finds an energy-efficient placement of handlers. MagnetOS applications are comprised of event handlers that communicate with each other by raising well-typed events. Event signatures specify the types of the arguments passed with the event, as well as the return type of the handler. By default, all externally visible entry points, such as methods in a Java object specification, serve as event declarations, and method bodies constitute the default handler for that event in the absence of overriding runtime event bindings. Consequently, the MagnetOS programming model closely parallels the Java virtual machine, providing access to standard Java libraries and enabling familiar development tools to be used to construct distributed applications.

Our MagnetOS implementation consists of a static application partitioning service that resides on border hosts capable of injecting new code into the network, a runtime on each node that performs dynamic monitoring and component migration and a set of policies to guide object placement at run time. The static partitioning service takes regular Java applications and converts them into distributed components that communicate via events 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.

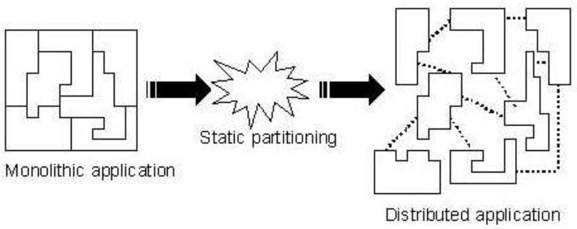

Figure 1: A static partitioning service converts monolithic Java applications into distributed applications that can run on an ad hoc network and transparently communicate by raising events.

The algorithms for event handler placement form the core of our system. We present practical, online algorithms for finding an energy-efficient distribution of application components in an ad hoc network (Figure 2). This paper examines the effectiveness of these algorithms in reducing energy consumption and extending system lifetime in the context of three application benchmarks, and examine their impact on system longevity. These 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 have implemented the system described in this paper and deployed it on x86 laptops, Transmeta tablets, and StrongArm PocketPC devices. Since current Java virtual machines place significant minimum memory requirements from their platforms, we have developed our own space-optimized JVM, suitable for “headless PocketPCs” which are cheap, have low energy requirements, and approximate what one might find in an embedded commodity device a few years from now. We report on our experience with developing and deploying applications in a physical testbed of 16 mobile nodes using this implementation. To increase the scale of our evaluation, we report results from simulation studies, validated against our implementation, which show that the MagnetOS system can achieve significant improvement in system longevity over static placement and standard load-balancing techniques.

This paper makes three contributions. It proposes a novel programming model for ad hoc networks where the entire network appears as a single unified system to the

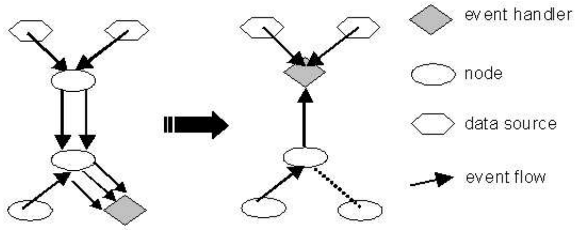

Figure 2: Migrating components closer to their data sources increases system longevity and decreases power consumption by reducing total network communication cost.

programmer. It describes the design and implementation of an operating system based on this model, where the distributed application is expressed at the language level as a single, cohesive application. The system operates by automatically partitioning applications and transparently migrating event handlers at runtime. Second, we propose practical, adaptive, online algorithms for finding an energy-efficient placement of application components in an ad hoc network. These algorithms form the central challenge to automatic energy-efficient execution in the system layer, and we demonstrate that limited, locally-collected information can lead to effective placement decisions. Finally, we demonstrate that these algorithms achieve high-energy utilization, extract low overhead, and improve system longevity.

In the next section, we describe related work on operating system support for ad hoc 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 within this environment. We summarize our contributions in Section 5.

2 Related Work

Past work has examined distributed operating systems, ad hoc networks, power management and programmability of ad hoc sensor network, though few systems have examined all of them.

2.1 Distributed Systems

Data and code migration have been examined extensively in the context of wired networks of workstations. Early landmark systems, such as V [9], Sprite [35], Accent [39], and LOCUS [37], implemented native operating system facilities for migrating processes between nodes on a tightly coupled cluster. Glunix [14] provides facilities for managing applications on a tightly connected networks of workstations. More recently, the cJVM [4] and JESSICA [32] projects have examined how to extend a Java virtual machine across a high-performance cluster. Others, including Condor [30] and CoCheck [43], 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 multihop 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 [24] provides transparent code migration for programs written in the Emerald language, where the migration is directed by source-level programmer annotations. Thor [29] provides persistent objects in a language-independent framework. It enables caching, replication and migration of objects stored at object repositories. These seminal 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.

The closest approach to ours are some recent systems that focused on how to partition applications within a conventional wired network. The Coign system [22] 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. The ABACUS system [1] 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. [25] propose using static analysis to select tasks that can be executed remotely to save energy. J-Orchestra [46] performs application partitioning via rewriting, leaving dynamic migration decisions under application control. Spectra [13] 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 [23] provides relocation and messaging services to facilitate the construction of mobile applications. The Mobiware [3] and DOMT [26] toolkits are targeted specifically for ad hoc networks and provide an adaptive QoS programming interface. XMIDDLE [52] 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 network 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, reactive and hybrid routing protocols seek to pick efficient routes by proactively disseminating or reactively discovering route information, or both. While some protocols, such as PARO [16] and MBLR [47], have examined how to make power-aware routing decisions, all of these routing algorithms assume that the communication end-points are fixed. Directed diffusion [19] 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 Operating Systems

Prior work has examined how to construct space-constrained operating systems for sensor networks. TinyOS provides essential OS services for sensor nodes with limited hardware protection and small amounts of RAM [20]. Maté [27] provides a capsule-based programming model for in-network processing on sensor nodes. Impala [31], the middleware layer of ZebraNet, introduces a system activity model that handles a mix of regular operations and irregular events for long-running mobile sensor systems. MagnetOS is complementary to these standalone systems, in that its system-wide abstractions can be built on top of the services they provide.

2.4 Programming Paradigms for Wireless Networks

There have been many research efforts to improve the programmability of wireless ad hoc and sensor networks. Hood [51] and Abstract Regions [50] simplify application development by providing high level abstractions which group together nodes with similar properties. Hood provides a low-level data sharing mechanism among neighboring nodes. Abstract Regions is similar to Hood but also provides data aggregation and resource tuning options. TAG [33] treats the sensor network as a database and performs data aggregation. Sensorware [6] proposes an active sensor framework which employs lightweight and mobile control scripts. By contrast, MagnetOS takes a system approach that makes the whole network appear as a single system image to applications and does not require programmers to master a new programming paradigm.

2.5 Power Efficiency

Previous work has also examined how to minimize power consumption within an independent host through various mechanisms ([36], [17], [11], [44], [7], [41]), including low-power processor modes, disk spin-down policies, adapting wireless transmission strength and selectively turning off unused devices. More recent work [2] improves the power management by exposing new interfaces to applications and 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.

Remote execution, i.e. moving costly computation to well-equipped nodes, has been used ([5], [18], [28]) to conserve the energy of mobile devices. Others ([8], [15]) have proposed improving energy efficiency by controlling node mobility. Some of this work shares the same insight as MagnetOS in that they also try to improve power efficiency by application partitioning and computation migration. However, MagnetOS does not require the aid of well-equipped nodes or the assumption that node mobility is controllable.

3 System Overview

MagnetOS provides a single system image programming model based on events. An event is an indication that a named piece of code should be executed in response to a system, sensor, or application-initiated occurrence. Events are typed and may carry arguments and return values; they are defined in MagnetOS as procedures. Events may be synchronous, in which case the event invoker blocks until termination of the handler, or asynchronous, in which case control returns immediately to the component that raised the event. Applications consist of a set of event handlers, and execution consists of a set of event invocations that may be performed concurrently. Every application receives an initial event to start its execution; as well, components of an application receive specific events for initialization. Device drivers, sensors, and other peripherals may define events suitable for applications to handle, and typically raise them asynchronously in application components. MagnetOS provides the necessary abstractions to specify applications composed of such events and event handlers, the static service required to partition such an application into its separate components, and a runtime that modifies the behavior of the application dynamically to optimize system goals such as energy usage.

MagnetOS applications are specified as regular Java programs which define component boundaries as well as provide well-typed event specifications. This enables the bulk of the application logic to be expressed using familiar Java syntax and semantics. The MagnetOS rewriting engine partitions monolithic applications and distributes their components across an ad hoc network. The MagnetOS runtime then coordinates the communication and migration of these application segments across the nodes in the ad hoc network in order for the newly distributed application to execute in a power-efficient manner. We discuss the static and dynamic components of the MagnetOS runtime in the following sections.

3.1 Static Partitioning

MagnetOS partitions applications based on programmer annotations, though, in the absence of annotations, object boundaries delineate event handlers. Consequently, the unit of mobility in MagnetOS is typically a Java object instance, which we use synonymously with event handler. This transformation at class boundaries preserves existing object interfaces, and inter-object invocations define events in MagnetOS. 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 previous work on application rewriting at ingress points [42]. 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 rerun a costly verifier on application components.

3.1.1 Application Partitioning for the JVM

Static partitioning takes original application classes, and from each class extracts an event handler, a dispatch handle, an event descriptor, and a set of event globals associated with the event handler.

An event handler is a modified implementation of the original class that stores the instance variables of the corresponding object. Each handler is free to move across nodes in the network. Dispatch handles, on the other hand, are remote references through which components can raise events. That is, dispatch handles are used to invoke procedure calls on remote event handlers residing on other nodes. Event raises through the dispatch handle are intercepted by the MagnetOS runtime and converted into RPCs. This indirection enables code migration. As an event handler moves, the event raises occurring through the corresponding event dispatch handles are tracked by the MagnetOS runtime and directed to the new location of the event handler. Event descriptors capture the event signatures that the original code exposes to the rest of the application.

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 event handler at that location. This operation returns a corresponding, properly initialized dispatch handle, which is then used in subsequent event raises. In addition, MagnetOS converts remote data accesses into events corresponding to accessor functions to read and write named locations. Similarly, it converts lock acquisitions and releases into centralized operations at the event handler. Finally, typechecking and synchronization instructions (check-cast, instanceof, monitorenter and monitorexit instructions, and synchronized methods) are rewritten to trap into the MagnetOS runtime.

The final component created for a class is a set of event globals. The event globals are static fields shared across all instances of an event handler. Each event handler retains pointers to the corresponding instance of event globals, and can therefore share state with other handlers.

3.2 Dynamic Object Management

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, migration, garbage collection, naming, and event binding.

3.2.1 Object Creation

In order to create a new instance of an event handler, an application contacts the local runtime and passes the requisite type descriptor and parameters for creation. The runtime then has the option of placing the newly created handler at a suitable location with little cost. It may choose to locate the handler 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 handlers are created locally. We chose this approach for its simplicity, and rely on our dynamic migration algorithms to find the optimal placement over time. Furthermore, short-lived, tightly scoped event handlers do not travel across the network unnecessarily. The application binaries, containing all of the 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 handler by calling its constructor and returns a dispatch handle.

Every object is tagged with a unique id, consisting of a (node id, object id) tuple, as well as a creation time derived from a Lamport clock kept independently at each node. The unique id can be generated without expensive distributed consensus, and the creation time enables object versions to be differentiated from each other, aiding in failure detection and recovery.

3.2.2 Remote Invocation and Migration

The runtime transparently handles invocations among the event handlers distributed across the network. Each runtime keeps a list of local event handlers. Dispatch handles maintain the current location of the corresponding handler, and processes raise events on behalf of application invocations by marshalling and unmarshalling event arguments and results.

MagnetOS can migrate both active and passive event handlers. A passive event handler consists purely of static data associated with an event handler; since no active computation is modifying the handler's associated data, it is migrated at run time by serializing handler state and moving it to a new node. In order to reduce the energy cost of migration, MagnetOS informs dispatch handles of relocation lazily, the next time they raise an event or the next time the garbage collector renews their object leases on remote objects. This notification is accomplished through forwarding references left behind when event handlers migrate. Chains of forwarding pointers, if allowed to persist for a long time, would pose a vulnerability - as nodes die, out-of-date event references may lose the path to the current location of the handler to which they are bound. MagnetOS collapses these paths whenever they are traversed. Periodic lease updates in lease-based garbage collection requires periodic communication between dispatch handles and event handlers, which provides an upper-bound on the amount of time such linear chains are permitted to form in the network.

Migrating active event handlers efficiently requires effectively capturing the current state of the computation being performed by that handler, and relocating it to a new node. Since this state may be large, migration of active event handlers is a costly operation. MagnetOS takes into account the cost of the migration when making a decision to relocate a handler. In some cases, such as a node whose battery level is below a critical threshold required to offload all event handlers at that node, active migration is invoked in order to preserve the computation without disruption. In order to provide the migration of active handlers without incurring excessive run time costs, the MagnetOS static partitioning service injects code to periodically check a flag in each basic block. If the handler has been identified for migration, the MagnetOS runtime sets this flag. When the rewritten code detects that the flag is set, it checkpoints the current state of computation and traps into MagnetOS runtime. MagnetOS runtime transports the state and resumes the computation at the destination. Compared to other approaches based on periodic checkpointing like JavaGoX[40] or state tracking on a parallel stack like Merpati[45], this approach, similar to Brakes[48], incurs the cost of object checkpointing only when the event handler needs to be migrated.

3.2.3 Handling Failures

Executing an application in a distributed setting introduces a new set of failure modes. Since attempting to preserve single-system application semantics in the presence of such failures is futile due to well-known impossibility results, MagnetOS provides reasonable defaults for common cases and exposes the unrecoverable errors to applications. The system provides traditional at-most-once semantics for event invocation; namely, events are guaranteed to execute at most once, and failure indications raised by the run time kernel are conservative.

The main insight guiding failure detection in the ad hoc domain with a single system image is that failure detection can be deferred until a node requires the results of a computation executed elsewhere. Consequently, frequent pings to check on the availability of other nodes, common in the wide area setting though costly from an energy perspective in an ad hoc network, can be avoided. MagnetOS performs health checks only when a node is waiting on the result of a computation from another node. The system uses keep-alive messages for long-running synchronous event invocations; the loss of a threshold number of consecutive keepalives indicate a failure to communicate between the two nodes, stemming from the MAC, transport, routing, or MagnetOS migration layers.

In response to such failures, MagnetOS attempts to mask as much as it can without violating application semantics. MAC layer failures, due to interference, congestion, or lack of connectivity, are retried at the transport layer. The routing layer is closely coupled to the transport layer and initiates a route discovery when triggered by the transport layer. The migration layer uses forwarding pointers to track objects that have moved, and when the pointer chain is broken due to a failure, falls back to a broadcast query for the target event handler as a last resort. Despite these measures, nodes that are disconnected may well experience unrecoverable errors.

Unrecoverable errors are reflected to applications as a special run time exception, one that can be implicitly raised by any procedure in the system, that notifies a caller that an event invocation has failed. The caller can catch such exceptions and take remedial action, taking care that the failure detectors are conservative; that is, an event may have been invoked, and the failure may well have occurred in notifying the invoker of the successful completion of the event. Each event invocation carries a unique identifier that can be used readily to guard against multiple invocations of a handler. In some cases, a network failure may have resulted in a portion of the application being broken off the rest of the network, along with corresponding application state. When the broadcast query for an event handler fails, the application is notified that such a failure has occurred. In response, applications that can tolerate some loss of state can regenerate new objects, while others can fail with a hard failure. The unique identifier and creation time attached to every object by its creating node allows applications to detect version mismatches. The migration algorithms described later keep track of the identifiers and version numbers of objects that contacted a given object within an epoch, thereby allowing the system to cheaply detect a mismatch subsequent to a network partition and recovery. Alternatively, applications can perform their own global disconnect recovery algorithms using the same object version information. In all cases, an exception is thrown and application specific recovery code supplied by the programmer can be invoked. Determining application semantics in order to recover automatically from complicated failures is a difficult problem; MagnetOS strikes a balance by asking for programmer direction in case of object loss and healed network partitions.

3.2.4 Explicit Overrides

The MagnetOS runtime provides an explicit interface by which application writers can manually direct component placement. This interface allows programmers to establish affinities between event handlers and ad hoc nodes. We provide two levels of affinity. Specifying a *strong* affinity between an event handler and a node effectively anchors the code to that node. This is intended for attaching event handlers 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 in order to ensure that MagnetOS applications provide at least as much control to the programmer as manually crafted applications, we do not advocate their use.

3.3 Event Handler Placement

In this section, we describe the algorithms we developed to automatically find an energy-efficient placement for application components.

All of our algorithms share the same basic insight: they shorten the mean path length of data packets by moving communicating objects closer together. They do 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 behavior. The placement decision is made locally, based on information collected during recent epochs at that node.

LinkPull 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.

PeerPull operates at the network level, and migrates components multiple hops at a time. In each epoch, PeerPull 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, PeerPull finds the host with which a given object communicates the most and migrates the object directly to that host.

NetCluster migrates components to a node in the most active cluster. Nodes that share the same next or last hop on the route are considered to be in the same cluster. At the end of each epoch, NetCluster finds the cluster that a object communicates with the most and then migrate the object to a randomly chosen node within that cluster.

TopoCenter(1) migrates components according to a partial view of the network built at each node, based on connectivity information attached opportunistically to each packet. Each node attaches its single-hop neighborhood to outgoing packets, subject to space constraints. The topology information is accumulated along the path to the destination. Nodes on the path extract the topology information from packets they forward. At the end of an epoch, each node computes the shortest paths between itself and other nodes in its partial network view and moves an object to the node such that the sum of migration cost (proportional to the shortest path) and estimated future communication cost (based on the packet statistics of the last epoch) is minimized. TopoCenter(Multi) is similar to TopoCenter(1), except that nodes attach all the topology information they know to outgoing packets.

All 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 balances out the communication load in the network. As a result, these algorithms can significantly reduce power consumption and improve the total system longevity for energy-constrained ad hoc networks.

3.4 MagnetOS API

MagnetOS provides several key abstractions for programmers to be able to implement distributed ad hoc networking applications effectively. These abstractions enable applications to name nodes and application components, to collect statistics and information on network and node behavior, and to set and direct migration policies. In addition, some existing Java abstractions are amended to reflect their new functionality and failure modes in the distributed MagnetOS setting. We describe the significant parts of the API below.

A **Node** encapsulates the notion of a physical host that is running a MagnetOS instance. It enables applications to name and query nodes. A node handle can be used to migrate application components, to anchor an object to a specific location in the network, and to query node properties such as link state, network topology information and energy status.

The **Link** abstraction refers to a physical link between two Nodes and the **NeighborSet** abstraction is the set of single hop neighbors of a given Node. These two abstractions enable applications to discover the network topology and link characteristics based on low-level information that is being updated by the operating system..

Energy enables applications to query the energy level of the underlying platform. Applications typically take actions in face of changing of battery energy. The Energy API allows applications to query the current energy level, the drain rate, and recharge frequency of the underlying heterogeneous platforms through a common interface.

A **Timer** abstraction enables applications to schedule events to be invoked at a predetermined time in the future. Timers can be migrated between nodes unless specifically pinned by the application on a specified node. A closure, saved along with the timer at the time of registration, provides the necessary context for event arguments when the timer fires.

Lock is a remote synchronization object, analogous to the monitor in Java. Standard monitor and condition variables can only be used locally. In MagnetOS, Lock acquisitions and releases are converted into remote operations on the event handler. Identifiers used for threads are replaced by globally unique identifiers for active event handlers, and keep-alives are added between the remote lock holder and the lock instance in order to detect network failures. In case of such a failure, a RemoteLockable object is restored to its state saved at the point of lock acquisition and the lock is released with an exception. In order to guard this scheme from leading to inconsistent application state in the presence of nested locks (i.e. where a failure rolls back the state of one, but not all objects), MagnetOS enables applications to override the acquisition and releasing methods. For instance, if a RemoteLockable object is stateless, the Lock acquisition method can be overridden such that no checkpointing is made, and if the recovery requires rollback of multiple objects, the restore method can perform the necessary cleanup.

A **Thread** in MagnetOS is an execution context that can perform a sequence of synchronous event raises. It differs from threads in standard Java in that it is distributed over multiple nodes and can migrate from one node to another while Java threads are confined to a single node. Each Thread in MagnetOS has a unique identifier, which is crucial for the remote synchronization mechanism in MagnetOS.

All of these abstractions simplify the application development, gives programmers more flexibility, and enables applications to take advantage of information already kept by lower software layers in the MagnetOS runtime.

3.5 Implementation Support for Ad hoc Networks

The ad hoc networking domain places additional constraints on the runtime implementation. First, multihop 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.

Second, standard communication packages such as Sun's RMI are designed for infrastructure networks, and are inadequate when operating on multihop ad hoc networks. Frequent changes in network topology and variance in available bandwidth require MagnetOS to migrate objects, requiring the endpoints of an active connection to be modified dynamically as objects move. We have built a custom RPC package based on a reliable datagram protocol [21] that allows us to easily modify the communication endpoints when components move and is responsible for all communication between dispatch handles and corresponding event handlers.

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 communication and records the source and destination for all incoming and outgoing events. MagnetOS 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, in practice 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.

4 Evaluation

In this section, we examine the power efficiency of automatic migration strategies in MagnetOS. We first evaluate the core automatic migration algorithms, LinkPull, PeerPull, NetCluster, TopoCenter(1), and TopoCenter(Multi), in three different benchmarks. We compare our algorithms with manual (Static algorithm), natural load balancing (Random algorithm), and optimal (See definition below), and show that they achieve good energy utilization, improve system longevity, and are thus suitable for use in general-purpose, automatic migration systems. Next, we report results from microbenchmarks to show that automatically partitioning applications does not extract a large performance cost. We also present evidence showing that the memory costs of a specially tuned Java virtual machine is within the resource-budget of next generation ad hoc nodes. Finally, we show through a physical deployment that the simulated results match those observed in the real world.

4.1 Benchmarks and Workload

We evaluated the performance and efficiency of MagnetOS event handler migration strategies in three representative applications, each with a unique communication pattern and application workload. The applications were chosen to span a wide range of possible deployment environments. We first describe the setup and workload for each application, then examine their performance under MagnetOS.

4.1.1 SenseNet

We first 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 a well-equipped central node, extracts high-level data on events from the condensers, and sends results to an external wired network.

The application is run on a 14 by 14 grid of sensors, each placed 140 meters apart with a uniformly distributed jitter of 50 meters. The communication and sensing radius is 250 meters. The grid is partitioned into four quadrants, and a single condenser is assigned to aggregate and process data for each quadrant. The workload consists of two bodies that move through the sensor grid in randomly chosen directions. We measure the total remaining energy across all nodes, sensor coverage, number of drained sensor nodes, number of nodes not reachable by the display, and overall system longevity. We define system failure as the point when half of the field is no longer being sensed by sensors connected to the display node.

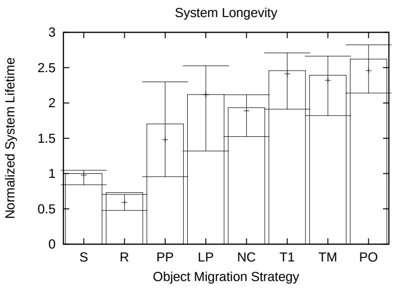

Figure 3: Automatic migration significantly extends system lifetime in the SenseNet application. Bars represent 25th and 75th quartiles.

4.1.2 Publish-Subscribe

Our second application consists of a basic publish-subscribe system. The application provides a channel abstraction to which clients can subscribe and publish. Channels act as mobile rendezvous points by accepting incoming messages and relaying them to each of the clients subscribed to the channel.

For this application, we generate a workload resembling a disaster recovery application. The workload consists of ten channels each with four subscribers. The four subscribers publish messages approximately every 10, 20, 30, and 40 seconds, respectively. We run the application on the same network layout as SenseNet. We measure total system throughput smoothed over 20 second intervals, number of nodes drained, and total remaining energy in the network. We stop the simulations when total throughput drops to zero during a 20 second interval.

4.1.3 FileSystem

Our final benchmark is a network file system that may be used in mobile ad hoc scenarios. This application consists of clients and files. Client objects are assigned to mobile devices, and access files over the network according to an external trace. File objects can reside on any node, and independently receive and process requests from clients.

This application is run on a randomly generated network with 196 mobile nodes, with approximately the same density as in SenseNet. The nodes move according to the random waypoint mobility model with a maxi-

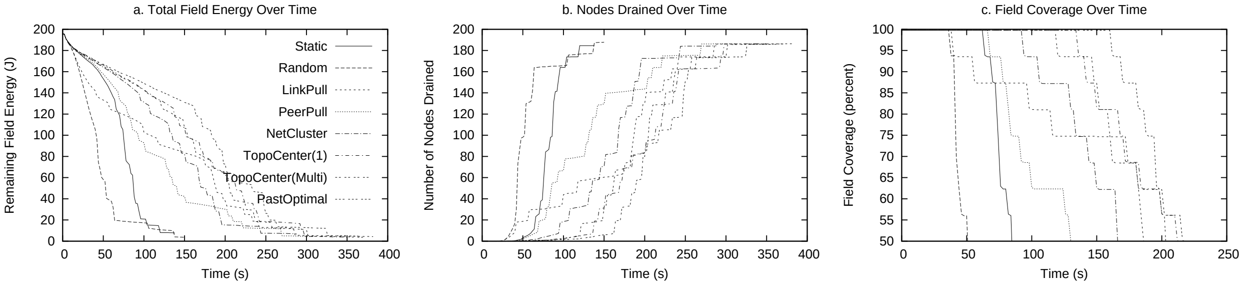

Figure 4: **SenseNet application:** Adaptive algorithms extract more energy out of the field and so increase field coverage and node availability.

mum node speed of 5 meters per second. The benchmark workload is based on the Auspex file system trace [10]. To compensate for the relatively limited capabilities of wireless nodes, we slow the trace by a factor of four. We measure the same statistics, and use the same stopping condition, as for the Publish-Subscribe application.

4.2 Simulation Methodology

We implemented a significant part of the MagnetOS system in **sns** [49], a scalable version of the Ns-2 network simulator. In order to accurately account for packet-level costs, we implemented a detailed energy model using parameters obtained from measurements of 802.11b wireless cards [12]. Computation costs are assumed to be negligible. We use the AODV protocol for wireless ad hoc routing, which includes support for both mobile and static node placements, and include the cost for route discovery, maintenance, and repair in our energy model. In all, we run each application with 16 scenarios each by varying the workload and network layout, averaging the results to obtain estimates of expected application behavior. For each application and scenario, we compare our algorithms described in section 3.3 with the following object placement and migration strategies:

Static Centralized corresponds to a static, fixed assignment of all movable objects to a single, central node in the network. All movable components remain at the home node for the entire duration of the simulation. **Random** selects a random neighbor as destination for each movable object at each epoch. It corresponds to a simple load-balancing algorithm, designed to avoid network hotspots. **PastOptimal** assumes that every node in the network has full knowledge of the network topology and moves objects to an optimal position based on the communication pattern in the last epoch. PastOptimal does not necessarily form the upper bound for other algorithms because it is still based on past behavior, and it may make bad decisions if the communication pattern changes significantly in the next epoch. Nevertheless, it is a good benchmark for migration algorithms based on past behavior of an object.

In addition to simulation-based evaluation, we implemented these benchmarks on top of our prototype system that supports x86/Windows, x86/Linux, and StrongArm/PocketPC platforms. The base system includes adaptive object placement policies, AODV ad hoc wireless routing, and automatic partitioning using Java bytecode rewriting. We will validate the simulation results against our physical test result later in this section.

4.3 Results and Discussion

The benchmarks represent a wide spectrum of different applications and communication models, and thus the relative performance of static and intelligent object migration policies varies with the application. Overall, these benchmarks show that the adaptive algorithms described above avoid hotspots in the network by moving objects intelligently. In addition, we find that the details of application communication and workload patterns impact the relative performance of different migration strategies, confirming the need for automatic and system-wide placement policies.

4.3.1 SenseNet

The SenseNet benchmark shows the clearest gains for the adaptive algorithms described above. Figure 3 shows that automatic migration increases system longevity by a factor of 2.5. This gain is achieved generally by moving objects away from hotspots and reducing mean packet distances. TopoCenter(Multi) performs a little worse than TopoCenter(1), which suggests that for this specific application, one-hop topology is good enough to make intelligent migration decisions and carrying multi-hop topologies will only increase the packet size and overhead, and decrease the system longevity.

Figure 4 shows SenseNet behavior in greater detail. The energy curves in (a) for the adaptive algorithms are more shallow than those for the Random and Static cases, which are steep and linear in the time of the simulation. Static suffers because of the energy bottleneck it creates around the fixed locations it has of system components. Random, a standard approach to distribute load, actually hurts energy performance by paying too much in migration costs.

The unreachable nodes (b) and coverage (c) graphs show two separate performance metrics with similar insights. Adaptive placement and migration save energy

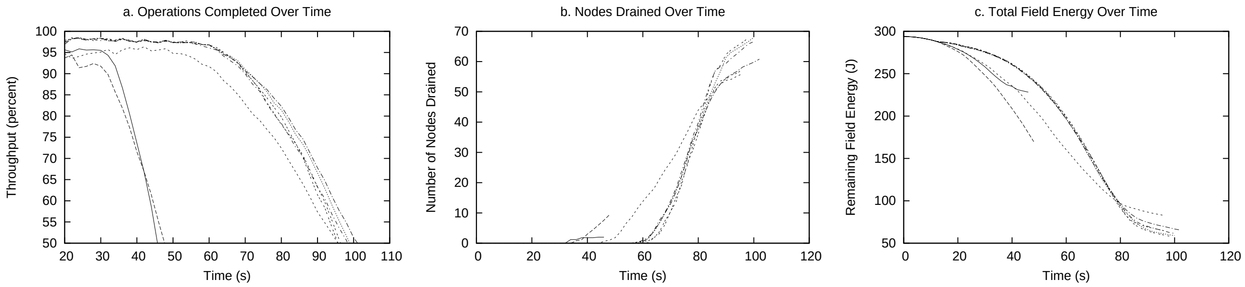

Figure 5: **Publish-Subscribe benchmark.** Adaptive algorithms improve throughput, availability and energy utilization.

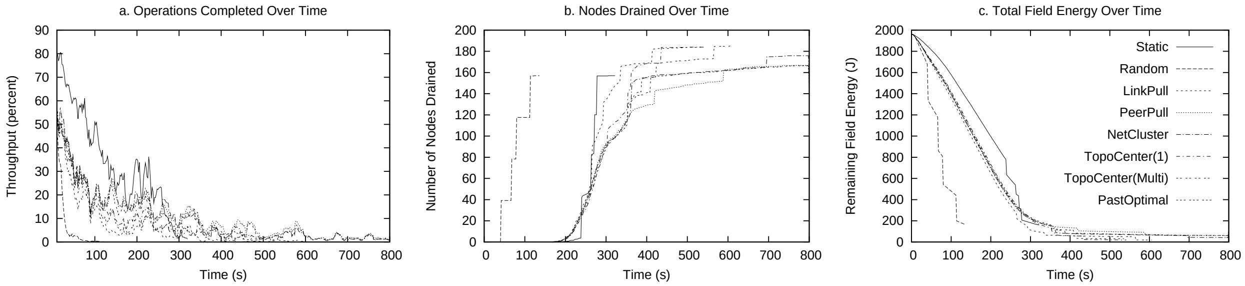

Figure 6: **FileSystem benchmark.** Comparison of adaptive algorithms and static and random load balancing.

and distribute load, which extends node lifetimes and increases longevity for the network.

4.3.2 Publish-Subscribe

The Publish-Subscribe application differs substantially from the SenseNet application. It consists of a relatively small number of rendezvous points, each communicating with a stable set of clients. This enables route discovery costs, incurred when objects are migrated, to be amortized over a large number of accesses.

Figure 5a shows that the Centralized approach fails very early because of the large hotspot in client accesses around the center of the network. We can see in each of the graphs that this approach initially achieves as high throughput as the dynamic migration strategies, but it dies early when it uses up the energy in the center of the network. The Random approach shows that randomizing the location of objects fails to achieve any savings, as it incurs the migration cost without the benefits of intelligently placing objects in the network.

MagnetOS policies do significantly better, since they place the rendezvous points near clients that access them frequently, reducing the ongoing cost of publish-subscribe operations. LinkPull does not do as well as the other adaptive algorithms because it requires several epochs to move the rendezvous points to the clients when subscriptions change. All other algorithms are very close to PastOptimal in their power performance.

4.3.3 FileSystem

In our file system benchmark, the centralized static algorithm performs very well initially, but fails afterwards with a steep cliff. The main reason for this is that Static avoids the energy cost for migration and route discovery but fails to avoid the hotspot around the central file server. Node mobility helps mitigate the hotspots somewhat and helps Static perform better in this benchmark than in the other two, but Static nevertheless suffers eventually. The lower throughput of adaptive algorithms initially is mainly due to changes of file locations. File operations following file migration require new route discovery, which increases latency and reduces the throughput.

The benefit of using an adaptive algorithm is clearly seen in the number of nodes drained (Figure 6b), where adaptive algorithms manages to spread the drain more evenly over the nodes of the network. The simple PeerPull algorithm performs surprisingly close to the PastOptimal algorithm. The Static and Random algorithms cannot do so intelligently and incur large losses of nodes due to hotspots.

Figure 6c shows similar results. The Static algorithm does well initially because it does not incur any costs for migration and route discovery. The centralized algorithm only has a single destination for all data flows. This layout is an optimal case for the AODV routing layer. All the adaptive cases, by event handler migration, have individual files which are located at many different points in the network. This leads to hundreds of different destinations, a load which is significantly more expensive for AODV to compute and maintain. Although adaptive algorithms consume more energy overall, that energy consumption is spread out over the whole network.

This benchmark shows that in applications where workload is comparatively low and routing cost dominates the overall energy consumption, adaptive algorithms can still help by distributing the energy cost throughout the network and avoiding the hotspots around central nodes.

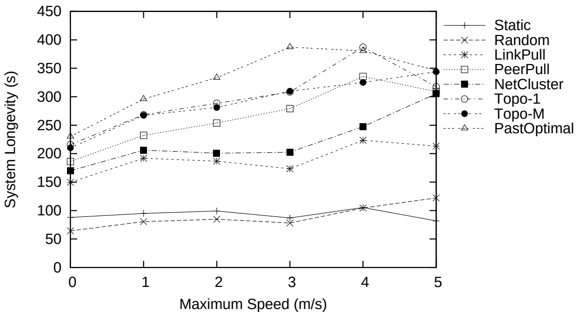

Figure 7: The average system longevity as a function of the maximum node speed.

4.3.4 Node Mobility

We evaluate the effect of node mobility on the performance of MagnetOS using the SenseNet benchmark. The simulation setup is modified to incorporate node movement according to the random waypoint mobility model with a pause time equal to $(50/S)$ seconds, where S is the maximum node speed. We vary the value of S and measure the average system longevity for each algorithm. Figure 7 shows that system longevity increases initially with mobility as physical movement acts as a simple load balancing scheme that avoids hotspots. Mobility also increases the route rediscovery cost and decreases the effectiveness of the prediction of our adaptive algorithms. However, adaptive algorithms perform well overall in mobile scenarios.

4.4 Validation

We validate the simulation results against our system implementation. We use the SenseNet benchmark for validation. Our implementation code executes on Linux 2.4.18 on Dell inspiron 2650 laptops, each equipped with Celeron M 1.5GHz processor, 256MB memory and Dell TrueMobile 1150 series 11Mbps Wireless LAN Adapters. To exclude the energy consumption of other devices like CPU, memory and LCD display, we change the wireless card driver to calculate energy cost of each packet transmission and reception.

The experiment setup is similar to the simulation setup we discussed above except we decrease the scale down to 16 nodes. 16 laptops are set up in a 4 by 4 grid, each running the MagnetOS runtime. The grid is partitioned into four quadrants, and a single condenser is assigned to aggregate and process data for each quadrant. Two objects move through the sensor grid in randomly chosen directions and speeds, triggering audio sensors on Magnetos nodes. In order to match the time scales of the physical experiment to the simulation, we measure the total energy cost of the field per event sensed by the sensor. We compare the results of the Static and PeerPull migration policies. Figure 8 shows that the energy consumption of the physical experiment closely match those of the simulations. The differences are attributable to variations in physical packet sizes from the average object size assumed in simulations.

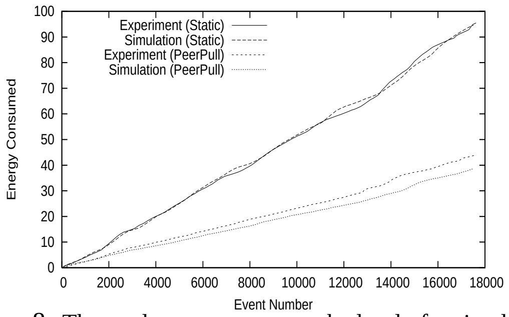

Figure 8: The total energy costs match closely for simulation and physical test results

Remote Call	Java RMI	MagnetOS
Null	403±16	172±6
Int	446±9	180±8
Obj. with 32ints	991±35	174±4
Obj. with 4int,2obj	884±21	177±7

Table 1: Remote method invocation comparison. All times in μs, average of 1000 calls.

4.5 Space and Time Overhead

An automatic approach to application partitioning and transparent object migration would be untenable if the performance of automatically partitioned applications suffered significantly. In the micro-benchmark shown in Table 1, 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, due partly to tight integration of system code with application code through binary rewriting.

Traditional Java VMs, such as Sun's JDK, support many features and are not optimized for space. Consequently, they require large amounts of memory and are not suitable for resource constrained nodes. For MagnetOS, we have developed our own JVM. and PocketPC/StrongArm devices. This JVM works on x86 laptops and PocketPC/StrongARM devices and performs space optimizations that include lazy class loading, constant pool sharing, stack and memory compression, and aggressive class unloading. Our JVM reduces the memory required to run MagnetOS from over 9 MB to approximately 1350 KB. This is well within the memory budget of existing mobile devices, and within a few years of Moore's Law growth of sensor nodes.

4.6 Summary

In this section, we examined three benchmarks built under the MagnetOS single system image model, and evaluated the performance of automatic migration strategies for energy-efficient execution. MagnetOS reduces energy consumption by actively moving communication endpoints and shortening the path packets traverse through the network. In turn, this reduces hotspots, increases energy utilization and extends system longevity. MagnetOS uses simple local metrics to make informed object placement decisions. In settings which exhibit locality, where active migration would shorten mean packet distances and yield energy savings, PeerPull, a local automatic migration policy, can adapt quickly, and find a good placement for objects. TopoCenter(1), which works by constructing a local view of the network topology and determining an optimal object destination within this view, is robust across applications and achieves performance close to PastOptimal.

5 Applications

We have developed various applications for the MagnetOS system. We report on these applications below; due to the high-level of abstraction provided by the MagnetOS system, developing these applications was simple, and the applications themselves are short and compact. We have deployed and run these applications on our physical testbed.

Audio Event Detection consists of sensors, condensers and displays, similar to the SenseNet application used in the simulations. Sensors are fixed on ad hoc nodes that are equipped with microphones. They monitor audio signals and send messages to condensers when the audio signal strength is above some threshold. Condensers can reside on any node. They aggregate audio data and filter noise. If enough events are detected within a time threshold, condensers notify the display of an object of interest in their monitoring area. The display is fixed on a well-equipped node and outputs the positions where audio events are detected. This application consists of 868 lines of Java code.

Rendezvous provides a rendezvous point for mobile nodes in an ad hoc network, similar to the type of service that may be set up by emergency response personnel. The rendezvous points are publish-subscribe spaces where clients can post and retrieve messages. They migrate from peer to peer in the network according to dynamic usage of the network and energy constraints. This application consists of 290 lines of Java code.

Video Multicast consists of Videosource, Framecarrier, and Videoclients. The VideoSource is fixed on an edge node that is equipped with a camcorder and captures video data. The VideoSource sends video frames to the FrameCarrier, which then multicasts the the video frame to all the clients. This application consists of 222 lines of Java code.

NetMixer builds a global sound processing engine in an ad hoc network that enables a large area to be monitored. It consists of three types of components: microphones, netmixers and speakers. Microphones are fixed on particular nodes, where they capture audio data and send them to the NetMixers. NetMixers combine the inputs from microphones into a single channel audio data using the channel-averaging technique used by soxmix[34]. Speakers combine data from all the NetMixers, produce new n-channel audio data (where n is the number of NetMixers), and play them back. This application consists of 838 lines of Java code, with most of the complexity in audio capture and playback.

Overall, it was trivial to develop distributed applications for a system that provided high-level features such as object migration, thread migration and energy efficient object placement. While these applications are not production-quality, they are nevertheless functional and illustrate how system support for mobility can simplify application development.

6 Conclusions

In this paper, we present the design and implementation of a novel operating system for ad hoc networks. Our system implements a network-wide, energy-efficient virtual machine on top of a collection of ad hoc 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 event handler creation, invocation and migration. We rely on a transparent RPC for node-independent communication between components. This distributed system defines a convenient programming model for ad hoc networks. This model provides the system with sufficient freedom to transparently move components in order to find an energy-efficient configuration.

We evaluate several algorithms for automatically determining where to locate application components in the network in order to minimize energy consumption. 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. In benchmarks with moderate to high locality of communication, automated migration can conserve power and achieve significant improvements in system longevity.

Ad hoc networking is a rapidly emerging area with few established mechanisms, policies and services. We hope that an operating system that makes the network appear as a single virtual machine, combined with system support for automatic migration, will create a familiar and power-efficient programming environment, thereby enabling rapid development of platform-independent, power-adaptive applications for ad hoc networks.

References

[1] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic Function Placement in Active Storage Clusters. In *Proc. of the USENIX Technical Conference*, San Diego, CA, June 2000.

[2] M. Anand, E. B. Nightingale, and J. Flinn. Ghosts in the Machine: Interfaces for Better Power Management. In *Proc. of MobiSys*, Boston, MA, June 2004.

[3] O. Angin, A. Campbell, M. Kounavis, and R.-F. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Personal Communications Magazine, Special Issue on Adapting to Network and Client Variability*, August 1998.

[4] Y. Aridor, M. Factor, and A. Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *Proc. of ICPP*, Wakamatsu, Japan, September 1999.

[5] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-Based Remote Execution for Mobile Computing. In *Proc. of MobiSys*, San Francisco, CA, May 2003.

[6] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In *Proc. of MobiSys*, San Francisco, CA, May 2003.

[7] L. S. Brakmo, D. A. Wallach, and M. A. Viredaz. uSleep: A Technique for Reducing Energy Consumption in Handheld Devices. In *Proc. of MobiSys*, Boston, MA, June 2004.

[8] S. Chakraborty, D. K. Y. Yau, and J. C. S. Lui. On the Effectiveness of Movement Prediction to Reduce Energy Consumption in Wireless Communication. In *Proc. of SIGMETRICS*, San Diego, CA, June 2003.

[9] D. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[10] M. D. Dahlin, C. J. Mather, R. Y. Yang, T. E. Anderson, and D. A. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proc. of SIGMETRICS*, Nashville, TN, May 1994.

[11] F. Douglis, P. Krishnan, and B. Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *Proc. of MLICS*, Ann Arbor, MI, April 1995.

[12] L. M. Feeney and M. Nilsson. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proc. of InfoCom*, Anchorage, AK, April 2001.

[13] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Proc. of HotOS*, Elmau/Oberbayern, Germany, May 2001.

[14] D. Ghormley, D. Petrou, S. H. Rodrigues, A. Vahdat, and T. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. *Software-Practice and Experience*, 9(28):929–961, 1988.

[15] D. Goldenberg, J. Lin, A. S. Morse, B. Rosen, and Y. R. Yang. Towards Mobility as a Network Control Primitive. In *Proc. of MobiHoc*, Roppongi Hills, Tokyo, Japan, May 2004.

[16] J. Gomez, A. T. Campbell, M. Naghshineh, and C. Bisdikian. PARO: Conserving Transmission Power in Wireless Ad hoc Networks. In *Proc. of ICNP*, Riverside, CA, November 2001.

[17] D. Grunwald, P. Levis, K. I. Farkas, C. B. M. III, and M. Neufeld. Policies for Dynamic Clock Scheduling. In *Proc. of OSDI*, San Diego, CA, October 2000.

[18] S. Gurun, C. Krintz, and R. Wolski. NWSLite: A Light-Weight Prediction Utility for Mobile Devices. In *Proc. of MobiSys*, Boston, MA, June 2004.

[19] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building Efficient Wireless Sensor Networks with Low-Level Naming. In *Proc. of SOSP*, Chateau Lake Louise, Banff, Canada, October 2001.

[20] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Network Sensors. In *Proc. of ASPLOS*, Cambridge, MA, November 2000.

[21] B. Hinden and C. Partridge. Reliable Data Protocol (RDP), Version 2. In *RFC 1151, IETF*, April 1990.

[22] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proc. of OSDI*, New Orleans, LA, February 1999.

[23] A. D. Joseph, A. F. D. Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proc. of SOSP*, Copper Mountain Resort, CO, December 1995.

[24] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM TOCS*, 6(1), February 1988.

[25] U. Kremer, J. Hicks, and J. Rehg. Compiler-directed Remote Task Execution for Power Management: A Case Study. In *Proc. of COLP*, Philadelphia, PA, October 2000.

[26] T. Kunz and S. Omar. A Mobile Code Toolkit for Adaptive Mobile Applications. In *Proc. of WMCSA*, Monterey, CA, December 2000.

[27] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proc. of ASPLOS*, San Jose, CA, October 2002.

[28] Z. Li, C. Wang, and R. Xu. Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme. In *Proc. of CASES*, Atlanta, GA, November 2001.

[29] B. Liskov, M. Day, and L. Shrira. Distributed Object Management in Thor. In *Proc. of the Workshop on Distributed Object Management*, pages 79–91, Edmonton, Canada, August 1992.

[30] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. *Technical Report 1346, University of Wisconsin-Madison*, April 1997.

[31] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proc. of MobiSys*, Boston, MA, June 2004.

[32] M. J. M. Ma, C.-L. Wang, F. C. M. Lau, and Z. Xu. JESSICA: Java-Enabled Single System Image Computing Architecture. In *Proc. of PDPTA*, Las Vegas, NV, June 1999.

[33] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. of OSDI*, Boston, MA, December 2002.

[34] L. Norskog. Sox - Sound eXchange. *http://sox.sourceforge.net/*.

[35] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[36] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proc. of SOSP*, Chateau Lake Louise, Banff, Canada, October 2001.

[37] G. Popek and B. Walker. *The LOCUS Distributed System Architecture*. MIT Press, Cambridge, MA, 1985.

[38] G. Pottie and W. Kaiser. Wireless Integrated Network Sensors. *Communications of The ACM*, 43(5):51–58, May 2000.

[39] R. F. Rashid and G. G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proc. of SOSP*, Pacific Grove, CA, December 1981.

[40] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Proc. of ASA/MA*, ETH Zurich, Switzerland, September 2000.

[41] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. D. Micheli. Dynamic Power Management for Portable Systems. In *Proc. of MobiCom*, Boston, MA, August 2000.

[42] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Proc. of SOSP*, Kiawah Island Resort, SC, December 1999.

[43] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proc. of IPPS*, pages 526–531, Honolulu, HI, April 1996.

[44] M. Stemm and R. Katz. Measuring and Reducing Energy Consumption of Network Interfaces in Handheld Devices. In *Proc. of MoMuc*, Princeton, NJ, September 1996.

[45] T. Suezawa. Persistent Execution State of a Java Virtual Machine. In *Proc. of the Java Grande Conference*, San Francisco, CA, June 2000.

[46] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proc. of ECOOP*, Málaga, Spain, June 2002.

[47] C. Toh. Maximum Battery Life Routing to Support Ubiquitous Mobile Computing in Wireless Ad Hoc Networks. *IEEE Communications*, June 2001.

[48] E. Truyen, B. Robben, B. Vanhuate, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proc. of ASA/MA*, Zurich, Switzerland, September 2000.

[49] K. Walsh and E. G. Sirer. Staged Simulation: A General Technique for Improving Simulation Scale and Performance. *ACM TOMACS, Special Issue on Scalable Network Modeling and Simulation*, April 2004.

[50] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proc. of NSDI*, San Francisco, CA, March 2004.

[51] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A Neighborhood Abstraction for Sensor Networks. In *Proc. of MobiSys*, Boston, MA, June 2004.

[52] S. Zachariadis, C. Mascolo, L. Capra, and W. Emmerich. XMIDDLE: Information Sharing Middleware for a Mobile Environment. In *ICSE, Demo Presentation*, Orlando, FL, May 2002.