TOWARDS A POWER EFFICIENT PROGRAMMING MODEL FOR AD HOC NETWORKS

Abstract

In this paper, we describe the design and implementation of a distributed operating system for ad hoc networks. The goal of our system is to extend total system lifetime for ad hoc networking applications through power-aware adaptation. We propose an event-based model for programming ad hoc networking applications. Our system automatically and transparently partitions applications into components and dynamically finds a placement of these components on nodes within the network to reduce energy consumption and increases the longevity. This paper describes our programming model, outlines the design and implementation of our system and examines automatic migration policies for ad hoc networks. 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, as well as the infrastructure on which they execute, 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. This lack of system support makes it difficult to develop ad hoc networking applications and to execute them in a resource-efficient manner.

Current state of the art in developing applications for ad hoc networks is to treat the network as a system of 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 that acts as a trusted arbiter between mutually distrusting applications allows conflicts to emerge between applications. For instance, in an acoustic sensor network where the primary application is to detect submarines and a secondary application is to track mammals, the system of systems model makes it difficult to express the relative priorities of the applications. A low priority application may interfere with higher priority applications simply by using a more aggressive migration policy that depletes the power supply of critical nodes in the system. 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 and outline a distributed operating system based on this model, called Flock. We show that the Flock approach can lead to increased energy efficiency for applications. 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 [Pottie & Kaiser 00]. Consequently, the goals of Flock are as follows:

- **Efficient:** 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.
- **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 an ad hoc 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.
- **Extensible:** The system should provide facilities for deploying, managing and modifying executing applications whose lifetime may exceed those of the network participants.
- **Compatible and Platform independent:** 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.

Flock meets these goals through an event-based programming model. Flock applications are structured as a set of interconnected, mobile *event handlers*, specified statically by the programmer as familiar objects in an object-oriented system. The Flock runtime, through application partitioning, distributes these event handlers to nodes in the ad hoc network, and, through dynamic migration, finds an energy-efficient placement of handlers within the network. Flock 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 event 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 Flock 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.

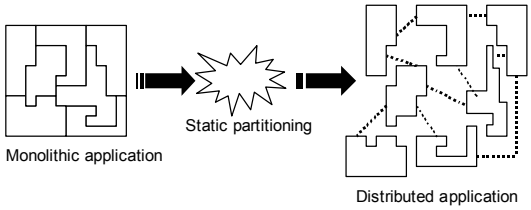


Figure 1: A static partitioning service converts monolithic Java applications into distributed applications that can run on an ad hoc network and transparently communicate by raising events.

Our Flock implementation 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 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.

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 built a prototype implementation that as well as a prototype implementation that runs on x86 laptops, Transmeta tablets, and StrongArm PocketPC class devices. We report results from simulation studies, which show that the Flock system can achieve significant improvement in system longevity over static placement and standard load-balancing techniques.

This paper makes three contributions. It proposes an event-based programming model for ad hoc networks that leverages existing language mechanisms for to specify distributed network programs. It describes the design and implementation of an operating system for ad hoc networks based on this model. This system operates by automatic 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. 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.

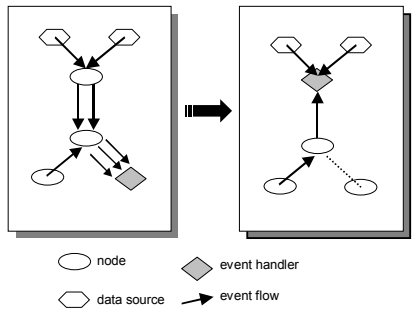


Figure 2: Migrating components closer to their data sources in an ad hoc network increases system longevity and decreases power consumption by reducing total network communication cost.

2. Related Work

Past work has examined distributed operating systems, ad hoc networks, and power management, though few systems have examined all three.

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 [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. Glunix [Ghormley et al. 98] provides facilities for managing applications on networks of workstations. 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, Flock 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. These seminal systems differ fundamentally from Flock 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 were some recent systems that 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. The ABACUS system [Amiri et al. 00] has examined how to migrate functionality in a storage cluster. Flock 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. Flock 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 [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. Flock 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 [Hill et al. 00]. Mate [Levis & Culler 02] builds on TinyOS to provide a capsule-based programming model for in-network processing on sensor nodes. Flock is complementary to these stand-alone systems, in that its system-wide abstractions can be built on top of the services they provide.

Previous 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

Flock provides an event-based programming model for ad hoc networks in three steps. First, an application is specified as a regular Java virtual machine program, defining component boundaries as well as well-typed event specifications. Next, this monolithic application is partitioned by a rewriting engine, distributing its functionality across the ad hoc network. The Flock 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 execute in a power-efficient manner. We discuss the static and dynamic components of the Flock runtime in the following sections.

3.1. Application Partitioning

The partitioning mechanism of Flock converts Java applications written and compiled for a single virtual machine into remote event handlers that can be dispersed and executed across an ad hoc network. This transformation enables the bulk of the application logic

to be expressed using familiar Java syntax and semantics.

Flock partitions applications based on programmer annotations, though, in the absence of annotations, object boundaries delineate event handlers. Consequently, the unit of mobility in Flock 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 Flock. 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 Flock 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 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 event handler. 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 Flock 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 Flock 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 Flock 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, Flock 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 (**checkcast**, **instanceof**, **monitorenter** and **monitorexit** instructions, and synchronized methods) are rewritten to trap into the Flock 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.

Flock provides a system abstraction similar enough to Java to facilitate easy programming and migration of existing applications. However, the Flock runtime is devoid of notions of threading, because they are ill suited to distributed computation. The Flock runtime replaces the notion of threads with an event-based model. Events in Flock are interruptible, independent computations. Application components communicate with each other by raising events. Raising events causes an event descriptor to be queued at the corresponding event handler. A set of worker threads at the appropriate node execute these events. The scheme decouples the description of a serial computation from the notion of a thread bound to a single processor. A dispatcher mechanism, similar to [Pardyak & Bershad 96], provides interposition and late binding.

We designed Flock as an event-based programming system based on Java for several reasons, each of which illustrates a conscious tradeoff in the design. Foremost, we chose Java because it is familiar to programmers, and allows programmers to easily and compactly express network-wide behavior. In contrast, a lower-level approach where programmers explicitly specify policies at the level of individual nodes or components would be more cumbersome than a system where such notions were expressed implicitly in the code. Our event-driven model is a departure from the traditional Java semantics of threads and concurrent execution. Threads are ill suited to computation in an ad hoc network because they explicitly tie computation to node resources such as stack frames and processors. An event-based model is efficient because the number of concurrent event handlers can be adjusted based on

the amount of memory and processing power at each node, without impacting the programming model. In addition, an event-based model decouples callers from callees and avoids making inter-node dependencies.

3.2. Migration Mechanisms

The Flock 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, event handler migration, garbage collection, naming, and event binding.

In order to create a new instance of an event handler, an application will contact the local runtime and pass 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.

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 process raised events on behalf of application invocations by marshalling and unmarshalling event arguments and results.

The Flock runtime implements a lease-based garbage collector for remote references to event handlers, with leases automatically renewed by live dispatch handles. As in RMI and Network Objects [Birrell et al. 94], our current implementation does not collect cycles in the reference graph. Local handlers are collected by the standard Java garbage runtime.

Flock migrates event handlers at runtime by serializing handler state and moving it to a new node. Dispatch handles are informed of the relocation lazily, the next time they raise an event or renew their leases. This is accomplished through forwarding references left behind when event handlers migrate. Long 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 not be able to trace a path to the current location of the event handler to which they are bound. Flock 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. A broadcast mechanism is used as a fallback to locate handlers by the identifier stored in a dispatch handle if the pointer-chain is broken due to a failure.

The Flock 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 Flock applications provide at least as much control to the programmer as manually crafted applications, we do not advocate their use.

3.3. Runtime Support for Ad hoc Networks

The ad hoc 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. Flock 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 multi-hop ad hoc

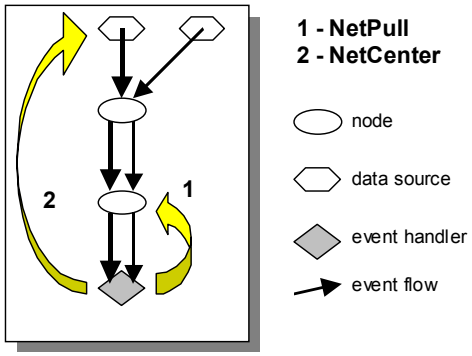


Figure 3: NetPull moves one hop towards the source of data whereas NetCenter moves directly to the source of most packets.

networks. Frequent changes in network topology and variance in available bandwidth require Flock 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 [Hinden & Partridge 90] 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 Flock 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. Flock 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 Flock uses these statistics to automatically migrate components.

3.4. Event Handler 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 balances 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.

4. Evaluation

In this section, we examine the power efficiency of automatic migration strategies in Flock. We first evaluate the core automatic migration algorithms, NetPull and NetCenter, in three different benchmarks, 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. Finally, we 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.

4.1. Benchmarks and Workload

We evaluated the performance and efficiency of Flock 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 Flock.

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 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 three 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 the display node.

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 14 by 14 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

Lastly, our final application 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 maximum node speed of 5 meters per second. The benchmark workload is based on the 1994 Auspex file system trace [Dahlin et al. 94]. 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 Flock system in Sns [Walsh & Sirer 03], 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 [Feeney & Nilsson 01]. 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 consider 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 mov-

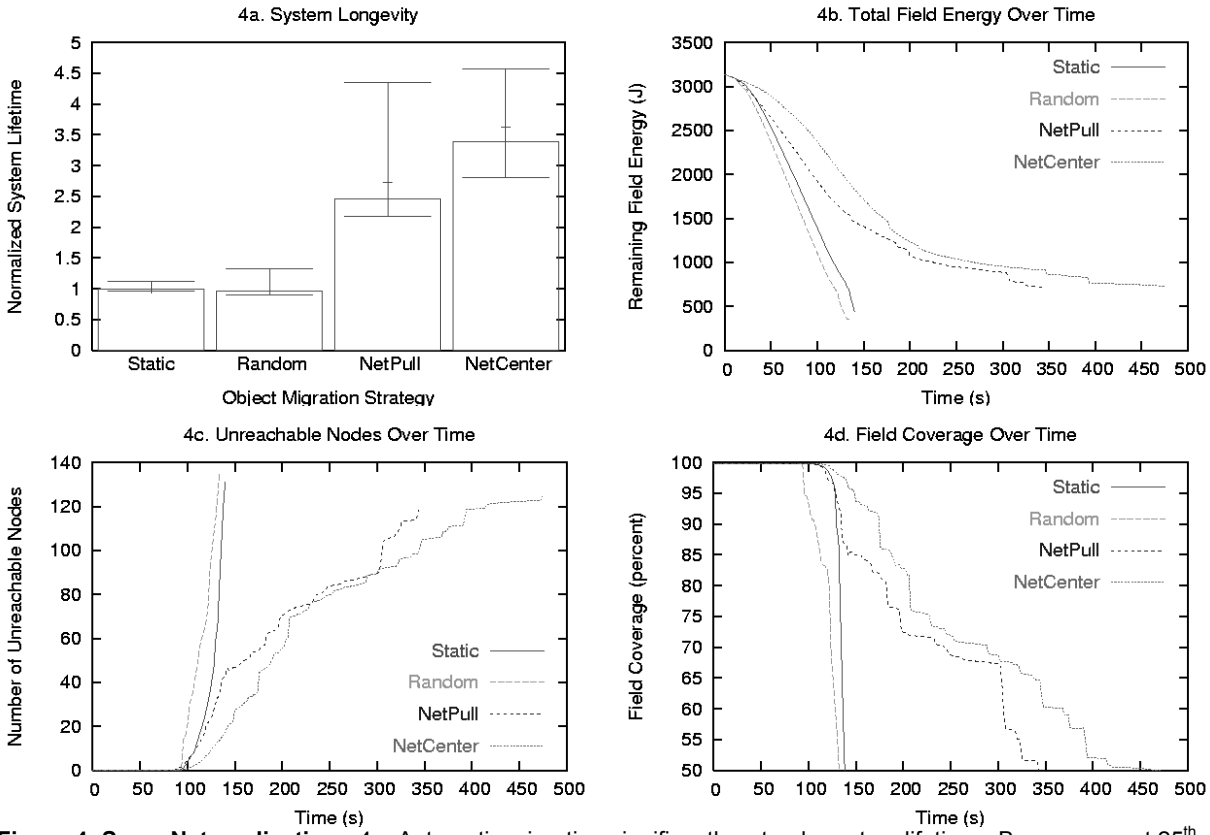


Figure 4: SenseNet application: 4a. Automatic migration significantly extends system lifetime. Bars represent 25th and 75th quartiles. 4b, c, and d show that adaptive algorithms extract more energy out of the field and increase the field coverage and node availability.

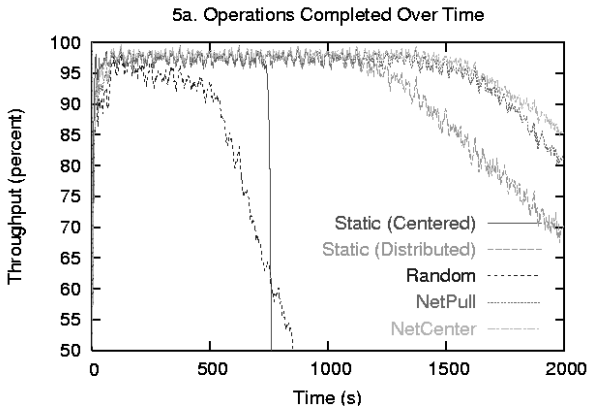
able components remain at the home node for the entire duration of the simulation.

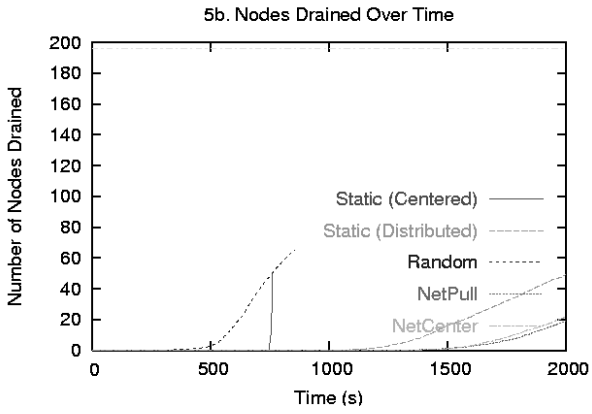
- Static Distributed corresponds to a static, fixed assignment of objects to nodes within the network. Movable objects are randomly assigned to nodes, and remain at those nodes for the entire duration of the simulation.
- Random selects a random neighbor as destination for each movable object at each epoch. It corresponds to a simple load-balancing algorithm, designed to avoid network hotspots.
- NetPull moves objects one hop along the most active adjacent communication link at each epoch to the most active neighbor.
- NetCenter moves objects directly to the node with which it communicated the most in the previous epoch.

In addition to simulation-based evaluation, we implemented these benchmarks on top of our prototype system that supports x86/Windows and StrongArm/PocketPC platforms. The base system includes adaptive object placement policies, AODV ad hoc wireless routing, and automatic partitioning using Java bytecode rewriting.

4.3. Results and Discussion

In the following sections, we examine each application benchmark in turn. 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.





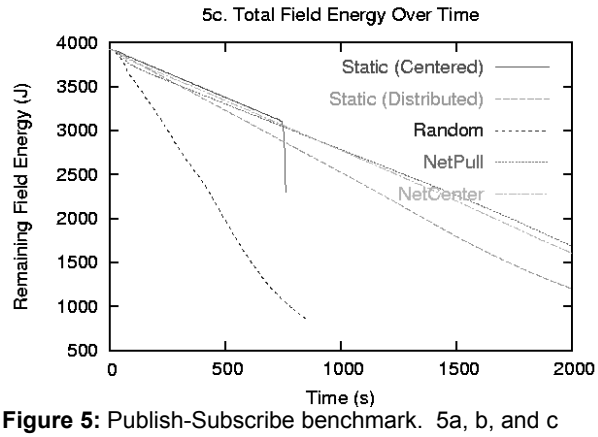
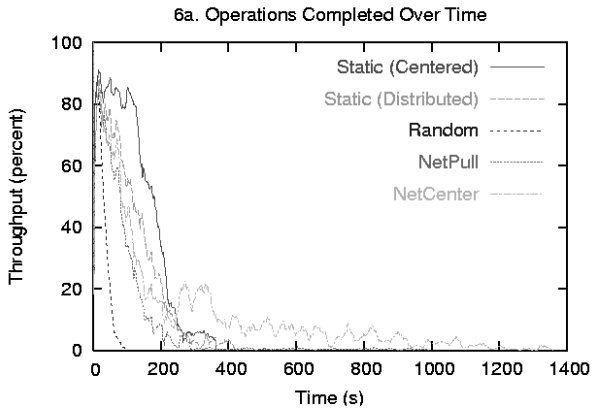
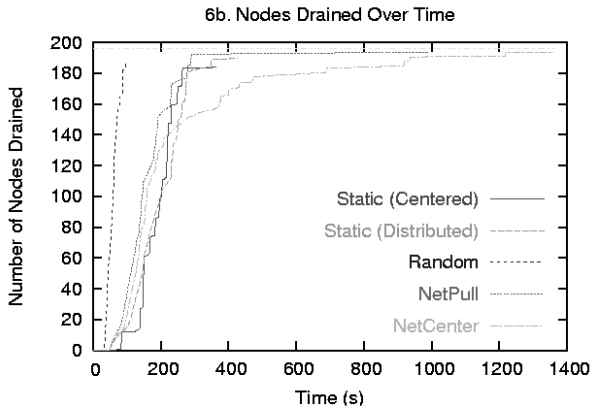


Figure 5: Publish-Subscribe benchmark. 5a, b, and c show that NetCenter and NetPull increase the longevity of nodes, the energy utilization, and the throughput.





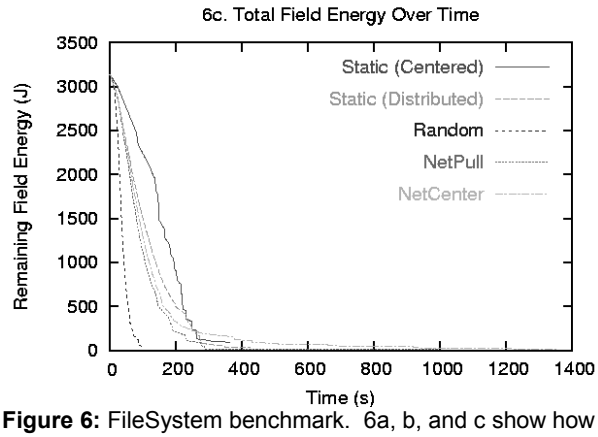


Figure 6: FileSystem benchmark. 6a, b, and c show how NetCenter and NetPull compare to static and random approaches.

4.3.1. SenseNet

The SenseNet benchmark shows the clearest gains for the adaptive algorithms described above. Figure 4a shows that automatic migration increases system longevity by a factor of 3. This gain is achieved generally by moving objects away from hotspots and reducing mean packet distances.

The energy graph in Figure 4b shows that the energy curves for NetCenter and NetPull 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 for system components. Random, a standard approach to distribute load, actually hurts energy performance by paying too much in migration costs. This benefit comes from the optimizing nature of the adaptive algorithms, which continuously try to find good placements.

The unreachable nodes and coverage graphs (Figure 4c and 4d) show two separate performance metrics with similar insights. Adaptive placement and migration save energy 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 commu-nicating 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 dynamic migration strategies, but it ter-minates early then it uses up the energy in the center of the network. The Random approach shows that ran-domizing the location of objects fails to achieve any savings or avoid hotspots, as it incurs the migration cost without the benefits of intelligently placing objects in the network.

The Distributed Static approach performs well, because it both avoids hotspots and does not incur object mobility costs. Adaptive policies do even better, since they place the rendezvous points near clients that access them frequently, reducing the ongoing cost of publish-subscribe operations.

4.3.3. FileSystem

In our file system benchmark, we discovered that the centralized algorithm performs better than any non-centralized one. This difference can be explained by the costs that each algorithm must pay to achieve the same level of availability. First, hotspots near the centralized server are mitigated because of node movement. Further, 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 distributed cases, by definition, have individual files which are located at many different points in the net-work. This leads to nearly 200 different destinations, a load which significantly more expensive for AODV to compute and maintain.

Among the non-centralized protocols, performance is dependent on and almost entirely determined by the cost of maintaining routes, since the per-file workload is typically very light. Many files, for example, are accessed only once every 60 seconds. The overhead associated with moving a file from one node to another, and the resultant cost of updating client routes to the file, outweighs the benefit of better placement in the network. This effect is exaggerated in proportion to the frequency of object movement. Figure 6a shows that policies that change object positions infrequently, such as Static Distributed, achieve better performance than those that perform frequent movement, such as Random. The adaptive algorithms do not perform as well in this context, since the highest costs are those of route maintenance. Figure 6c similarly shows that the energy costs of object mobility are not recompensed in this scenario. The only benefit of using an adaptive algorithm is seen in the number of nodes drained (Figure 6b), where NetCenter manages to spread the drain more evenly over the nodes of the network. The Static and Random algorithms cannot do so intelligently, and so have steep cliffs at which time many of the nodes die at once.

This benchmark shows that the value of the adaptive algorithms is seen in applications with relatively high load, and that when the application is simple, a static distribution of objects suffices. The long tails of the graphs are results of object motion: a node accesses a particular file frequently, and so the file is moved to that node. The communication between file and client can then proceed despite loss of connectivity.

4.4. 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 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, due partly to tight integration of system code with application code through binary rewriting.

Remote call	Java RMI	Flock
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.

The applicability of a Java-based system is limited by the ability of ad hoc network nodes to support the requisite services of a Java VM. Traditional virtual ma-

chines, such as Sun Java JDK 1.4 support many features and are not optimized for space. Consequently, they require large amounts of memory and are not suitable for ad hoc networks. For Flock, we have developed our own JVM for x86 laptops, and PocketPC/StrongArm devices. This JVM performs space optimizations including lazy class loading, constant pool sharing, stack and memory compression, and aggressive class unloading. Consequently, we reduce the minimum required memory to run Flock to approximately 1350 KB from over 9 MB. This is well within the memory budget of existing mobile devices, and within a few years' Moore's Law growth of sensor nodes.

4.5. Summary

In this section, we examined three benchmarks built under our event-based model, and evaluated the performance of automatic migration strategies for energy-efficient execution. Flock 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. Flock 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, NetCenter, a local automatic migration policy, can adapt quickly, and find a good placement for objects.

5. Conclusion

In this paper, we present the design and implementation of a event-based operating system for ad hoc networks. Our system implements a parallel, event processing engine 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 simple, local algorithms for automatically determining where to locate application components in the network in order to minimize energy consumption. Combined, these algorithms enable Flock 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. 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 high-level abstractions, such as an event-based programming model, 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

[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.

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

[Dahlin et al. 94] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Yang, Thomas E. Anderson, and David A. Patterson. A quantitative analysis of cache policies for scalable network file systems. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. 150–160. Nashville, Tennessee, 1994.

[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.

[Feeney & Nilsson 01] Laura Marie Feeney and Martin Nilsson. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Network-ing Environment. IEEE InfoCom. Anchorage, Alaska, 2001.

[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.

[Ghormley et al. 98] D.P. Ghormley, D. Petrou, S. H. Rodrigues, A.M. Vahdat, T.E. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. Software-Practice and Experience, 9 (28), 1998, pp. 929-961.

[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.

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

[Hill et al. 00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister. System architecture directions for network sensors, ASPLOS 2000, Cambridge, November 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.

[Levis & Culler 02] Philip Levis, David Culler. Mate: A Tiny Virtual Machine for Sensor Networks. ASPLOS, 2002.

[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.

[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.

[Pardyak & Bershad 96] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In Proceedings of OSDI. Seattle, WA, October, 1996.

[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.

[Walsh & Sirer 03] Kevin Walsh and Emin Gün Sirer. Staged Simulation for Improving the Scale and Performance of Wireless Network Simulations. Winter Simulation Conference. New Orleans, Louisiana, 2003.

[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.