AUTOMATIC CODE PLACEMENT ALTERNATIVES FOR AD-HOC AND SENSOR NETWORKS

Emin Gün Sirer Rimon Barr T. W. Danny Kim Ian Yee Yan Fung

Computer Science Department
Cornell University
Ithaca, NY 14853
{egs, barr, twkim, ifung}@cs.cornell.edu

August 31, 2001

Developing applications for ad-hoc and sensor networks poses significant challenges. Many interesting applications in these domains entail collaboration between components distributed throughout an ad-hoc network. Defining these components, optimally placing them on nodes in the ad-hoc network and relocating them in response to changes is a fundamental problem faced by such applications. Manual approaches to code and data migration are not only platform-dependent and error-prone, but also needlessly complicate application development. Further, locally optimal decisions made by applications that share the same network can lead to globally unstable and energy inefficient behavior.

In this paper we describe the design and implementation of a distributed operating system for ad-hoc and sensor networks whose goal is to enable power-aware, adaptive, and easy-to-develop ad-hoc networking applications. Our system achieves this goal by providing a single system image of a unified Java virtual machine to applications over an ad-hoc collection of heterogeneous nodes. It automatically and transparently partitions applications into components and dynamically finds a placement of these components on nodes within the ad-hoc network to reduce energy consumption and increase system longevity. This paper outlines the design of our system and evaluates two practical, power-aware, online algorithms for object placement that form the core of our system. We demonstrate that our algorithms can increase system longevity by a factor of four to five by effectively distributing energy consumption, and are suitable for use in an energy efficient operating system in which applications are distributed automatically and transparently.

Supported in part by ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

1 Introduction

Ad-hoc networks, formed through the collaboration of intelligent mobile nodes over wireless links, 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. Many interesting applications, however, entail collaboration between components distributed throughout an ad-hoc network. For example, sensor networks are often composed of three types of components: sensors acting as data sources, information consumers operating as data sinks, and numerous filters in-between for performing application-specific data processing. While the data sources and sinks may be coupled tightly to the nodes to which they are attached, there is often a high degree of freedom in the placement of the data processing components. This freedom, coupled with the dynamic environment posed by ad-hoc networks, both enables adaptive applications and makes it difficult to find the optimal distribution of application components among nodes.

Adapting to dynamically changing conditions by changing the distribution of functionality across a network is critical for many distributed ad-hoc networking applications [Satyanarayanan 96]. For example, the resources available to components at each ad-hoc node, in particular the available power and bandwidth, may change over time and necessitate the relocation of application components. Further, event sources that are being sensed in the external environment, such as tracked objects or chemical concentrations, may move rapidly, thereby shift network loads and require applications to adapt by migrating components. Finally, an application's behavior might change, as in the transition from defensive to offensive mode in a battlefront application, modifying its communication pattern and necessitating a reorganization of its deployed components within the network. Currently, ad-hoc networking applications either rely on a static assignment of components to nodes or use *ad-hoc*, manual policies and mechanisms for migrating code and data in response to change. A static assignment of functionality to nodes simplifies application design by obviating code migration and reduces meta-traffic in the network by eliminating component mobility, but it also leads to non-adaptive, fragile and energy-inefficient systems. The overall application will stall as soon as the critical nodes on the dataflow path run out of power or move out of transmission range. Manual approaches to code and data mobility suffer from being hard-to-develop, error-prone and platform-dependent. Each application using this approach needs to reimplement the same migration, monitoring and communication mechanisms, correctly, on every platform. Further, locally optimal policies pursued by applications may lead to globally unstable and energy-inefficient behavior when multiple applications share and interact on the same ad-hoc network. In essence, this approach suffers from building on an abstraction-level that is too low. A high-level operating system that provides the requisite mechanisms and policies for code mobility would not only simplify application development, but also ensure the integrity of system-wide goals in the face of multiple applications competing for resources.

In this paper, we outline the design of a single system image operating system for ad-hoc networks. The goals of our system are to enable the construction of applications with the following properties:

- ***Adaptive***: Applications should, with minimal effort, be able to respond to changes in their environment, their communication pattern, and the availability of resources in the network.
- ***General purpose***: Applications should be able to run on both ad-hoc and fixed networks. Porting an existing monolithic application to execute efficiently on an ad-hoc network should require little effort.
- ***Platform independent***: Applications should be able to execute on ad-hoc networks of heterogeneous nodes.
- ***Efficient***: Policies and mechanisms used for adaptation in the systems layer should not require excessive communication or power consumption. The default policies and mechanisms should yield good power utilization and maximize total system lifetime.

Our operating system meets these goals by providing the illusion of a single, unified Java virtual machine over an ad-hoc network of heterogeneous, physically separate, mobile hosts. Our system 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 RMI 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.

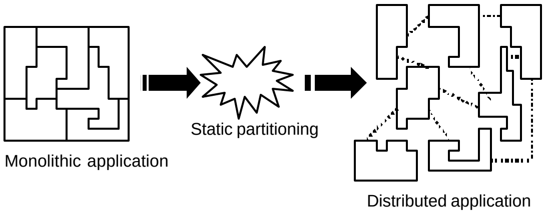

Figure 1: A static partitioning service converts monolithic Java applications into distributed applications that can run on an ad-hoc network and transparently communicate via RMI.

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

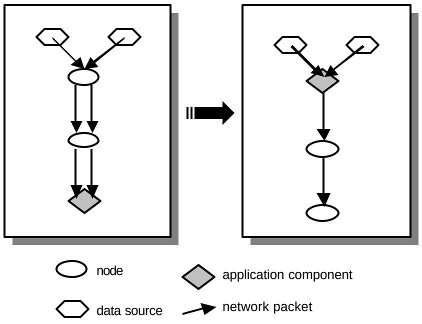

Figure 2: Migrating components closer to their data sources in a sensor network increases system longevity and decreases power consumption by reducing total network communication cost.

This paper makes three contributions. It outlines the design and implementation of a single system image operating system for ad-hoc networks, where the entire network appears to be a large Java virtual machine to applications, whose components are partitioned among the nodes automatically and migrated transparently. Secondly, we propose two practical, adaptive, online algorithms for deciding when and where to move application components. Finally, we demonstrate that these algorithms achieve high-energy utilization, extract low overhead, and improve system longevity, and are thus suitable for use in an operating system for transparent, automatic object migration.

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 *NetPull* and *NetCenter* within this environment. Section 5 describes our plans for future work. We summarize our contributions and the results of our simulations in Section 6.

2 Related Work

Prior research on ad-hoc networks has concentrated on ad-hoc routing algorithms such as DSR [Broch et al. 98b], AODV [Perkins 97], ZRP [Haas & Pearlman 98], TORA [Park & Corson 98], GeoTORA [Ko & Vaidya 00], PARO [Gomez et al. 01], OLSR [Jacquet et al. 98], LANMAR [Pei et al.

00], AMRIS [Wu & Tay 99] and others [MANET 01, Broch et al. 98a]. These routing algorithms move data from a source to a given destination(s) as efficiently as possible. They assume that the application was structured *a priori* by the programmer to operate optimally in an ad-hoc environment and that the communication pattern of the application is fixed. Our system complements routing algorithms to move application code, as well as data, around the network, radically altering the communication pattern of the overall application by bringing application components closer to the data sources and thereby providing increased system and application longevity.

Others have experimented with single system image (SSI) operating systems for wired networks of workstations. Sprite [Ousterhout 88], V [Cheriton 88], Ameoba [Steketee 95], Accent [Zayas 87], and LOCUS [Popek & Walker 85] implement native operating system facilities for migrating processes between nodes on a tightly coupled cluster. Most recently, the cJVM [Aridor et al. 99] and JESSICA [Ma et al. 99] projects provide a similar Java virtual machine-based, single system image to their applications. Others, including Condor [Litzkow et al. 97], libckpt [Plank et al. 95] and CoCheck [Stellner 96], provide user-level mechanisms for process migration without operating system support. These projects target high-performance, well-connected clusters, and face an entirely different set of challenges; the main goals of these SSI systems were load balancing and performance in a local area network for interactive desktop programs or compute-intensive batch jobs. In contrast, our system targets wireless multi-hop networks, where utilizing power effectively and maximizing system longevity is more important than traditional application performance.

Some recent systems have examined how to spatially partition applications (distribute application components) within a 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 office applications via a two-way minimum cut based on summary application profiles collected on previous runs. Extending this work, the ABACUS system [Amiri 00] has shown that incorporating dynamic information into component placement decisions can further improve application performance. Our work shares the same insight as Coign, in that we are also interested in the automatic partitioning and 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. We differ from both these systems with our focus on power conservation and system longevity through dynamic adaptation.

Various mobile computing projects have looked at how to structure mobile applications. They differ fundamentally from our system in that they do not provide an encompassing operating system platform, require explicit programmer control to trigger code migration, do not support an ad-hoc network model and target traditional applications. The Emerald system [Jul et al. 88] provides transparent code migration for code components written in the Emerald language, where code migration is directed by source-level programmer annotations in Emerald. The xDU project [Gehani 97] provides a similar framework for explicitly partitioning applications into distributable units using code annotations in Java. Legion [Lewis & Grimshaw 95] is a language independent, scalable, object-oriented operating system for wide-area infrastructure networks.

Other mobile computing projects have focused on creating mobility toolkits. The Rover toolkit [Joseph et al. 95] provides relocation and messaging services to facilitate the construction of mobile applications. More recent work includes the creation of toolkits specifically for ad hoc environments and applications. The Mobiware toolkit [Campbell 98] provides an adaptive-QoS programming interface, which assists in the management of object flows. XMIDDLE [Mascolo 01] assists with data management and synchronization. Our research takes a systems approach instead of a programmer driven toolkit and automatically manages the shared network and energy resources among ad hoc sensor applications. A benefit of this approach is that it obviates the need for applications to be recoded against a specific toolkit interface; applications are executed without modification atop our operating system. Furthermore, this approach ensures that applications, regardless of which toolkits they use, to behave in a cooperative manner.

Active networks [Tennenhouse & Wetherall 96] are a related technique for distributing computation within a network on a per-packet or per-flow basis. Our system differs from active networking efforts in that it offers higher-level, encompassing operating system services for ad-hoc applications. Its resource accounting is based on computational nodes and application components instead of packets or flows. It is, however, complementary to active networks, in that it may use an active network as a low-level mechanism for code distribution.

Some mobile applications can naturally be structured as reductions to distributed leader election. Much prior work has investigated the problem of selecting a leader node among a set of replicas [Chang and Roberts 79], [Garcia-Molina 82], [Awerbuch 87], [Ostrovsky 94], [Sayeed et al. 95], [Brunekreef et al. 96], [Singh 96], [Russell et al. 99], including recent work in an ad hoc network setting [Malpani et al. 00]. However, these algorithms do not account for the cost of keeping the state of replicas consistent, nor the effect of the messages exchanged for leader election on power consumption and system longevity. For applications with stateful components these costs and effects can be substantial.

Lastly, prior work has examined how to minimize power consumption within an independent host by manipulating the processor clock speed [Grunwald et al. 00, Weiser et al. 94]. 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

In this section, we briefly outline the design of our system, present our application partitioning and migration model, and discuss two algorithms for automatic, transparent object migration. A full discussion of the details of our single system image operating system and its implementation are beyond the scope of this paper. We will instead focus on the core of our operating system as it relates to object mobility, and will propose and evaluate two alternative algorithms for deciding when and where to move application components in ad-hoc networks.

3.1 Application Partitioning

Our code partitioning service operates by dividing Java applications into components that can migrate and execute over a network. This division is performed statically along object interfaces at class granularity[1]. Thus, the transformation preserves class interfaces, retains type safety, and works at the bytecode level without requiring source access.

Partitioning applications involves two separate modifications for object creation and invocation, respectively. Object creation instructions are replaced by calls to the local runtime, which is responsible for selecting an appropriate node and creating a new instance of the object on that node. This operation returns a handle to the remote object, which can then be used for subsequent object invocations. Object invocations are modified to go through an RPC mechanism [Birrell & Nelson 84] to invoke remote objects. We use the Java RMI interface for remote object invocation [Harold 00].

Our approach to partitioning applications statically is patterned after the Distributed Virtual Machine paradigm [Sirer et al. 99], and provides three advantages. First, the complex partitioning services need only be supported at code-injection points, and can even be performed offline. Second, since the system operation and integrity do not depend on the partitioning technique, users can partition their applications into arbitrary components if they so choose. Finally, this technique provides a convenient, default mechanism for transitioning legacy, monolithic applications to execute over ad-hoc networks.

---

[1] Consequently, application components correspond to Java objects in our system, and we use the two terms interchangeably in the rest of this paper.

3.2 Runtime Mechanisms for Object Migration

The runtime provides the dynamic services to facilitate communication between application components. It is invoked in response to three events: object creation, invocation, and migration. In addition, it transparently monitors application behavior to enable intelligent migration decisions.

In order to create a new instance of an object, an application contacts the local runtime and provides the type of the object to create. The runtime then has the option of placing the newly created object at a suitable location with little cost. It may choose to locate the object on the local node, at a well-known node or at its best guess of an optimal location within the network. In our current implementation, all new objects are created at the central node that connects the ad-hoc hosts to the wired network. We chose this approach for its simplicity, and rely on our dynamic object migration algorithms to find the optimal placement of objects over time. The application binaries, containing all of the object constructors, are distributed to all nodes at the time that the application is introduced into the network. Once created, the runtime simply initializes the object by calling its constructor and returns a remote handle to the object.

The runtime handles object invocations by locating the appropriate stubs, marshaling arguments, and performing the right RPC invocation via RMI. Our runtime independently keeps track of object locations in the OS layer. But we assume the presence of a standard ad-hoc routing protocol, like AODV or DSR, below our runtime to provide message routing. When an object migrates, it sends a multicast message to all of the components that have outstanding handles for itself and informs them of its new IP address. This multicast is tightly integrated with the underlying routing protocol to inform remote stubs of the new location of the object while simultaneously establishing a live route. In the case of AODV, the multicast packet contains a piggybacked RouteReply and causes intermediate nodes to acquire a distance metric and route for the new destination node. [*Note to reviewers: Our runtime implementation does not currently perform this optimization; however, we simulate its effects in the evaluation section*].

Finally, the runtime has a background thread that listens to incoming migration requests, receives new objects over the network and instantiates them.

3.3 NetPull and NetCenter

In the preceding sections, we described how our system provides object mobility. In this section, we describe two algorithms, named *NetPull* and *NetCenter*, which use these mechanisms to increase system longevity.

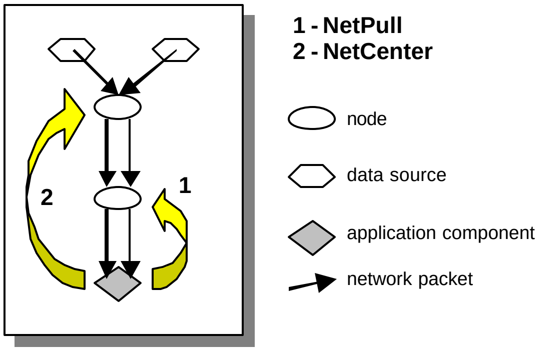

Figure 3: NetPull moves one hop towards the source of data whereas NetCenter moves directly to the source of most packets

Both *NetPull* and *NetCenter* share the same basic approach. 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. Each packet from a neighboring host is assigned a net force of one unit in the direction of that host. At the end of an epoch, the runtime examines all of the forces on the object and moves it one hop in the direction of net pull.

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 RMI invocations, and the destination addresses of outgoing RMI invocations for each object. The network source and destination addresses are often part of the packets sent during the RMI operation. For instance, they are often part of the IP source and destination fields, and thus can be collected without placing an extra 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, skipping over any intermediate nodes.

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

4 Evaluation

In this section, we evaluate *NetPull* and *NetCenter,* and show that they achieve good energy utilization, improve system longevity, and are thus suitable for use in a general-purpose, automatic, transparent object migration system. Below, we briefly describe our networking model, benchmark application, and simulation framework, and then discuss the results in detail.

4.1 Network Model and Benchmark Application

Our system targets general-purpose ad-hoc networks. These networks consist of geographically distributed nodes communicating over wireless links. In our simulations, all nodes have the same communication radius and they are connected to the fixed networking infrastructure via a single, centrally placed node. Each node initially stores a fixed, finite amount of energy. Sending a packet between any neighboring nodes exacts a constant communication cost, and the cost of local computation on a host is negligible in comparison.

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

4.2 Simulation Framework

We developed a fast, packet-level, statically parameterizable ad-hoc networking simulator in order to simulate large networks. In the style of ns2 [McCanne & Floyd 01], the simulator accounts for all communication costs, including AODV routing and route repair overhead. The simulator models the movement of every unicast and multicast packet and properly incurs the cost of moving a condenser and notifying all its sensors of the new location. We initialize the simulator with a uniform distribution of nodes on a plane, and vary parameters such as noise levels, field size, density, battery power, and communication and sensing radii. Sensing events are generated at random locations on the field, and with random durations and velocity vectors. Sensors that are in range detect these signals and generate application events. They can also, with small probability, generate fictitious events due to sensor noise.

4.3 Algorithms

We compare four different algorithms for transparent object migration:

- ***Static*** corresponds to a static, fixed assignment of objects to nodes within the network. Our components remain at the home node for the entire duration of the simulation.
- ***Random*** is designed to overcome a fundamental shortcoming of static placement, namely that the entire system can no longer function once key nodes are unreachable, by moving components around at random.
- ***NetPull*** looks at link-level data from its neighbors and moves to the most active neighbor.
- ***NetCenter*** keeps tracks of message sources and moves directly to the node with greatest activity.

4.4 Simulation Parameters

A simulation of a complex system such as this requires many parameters. We summarize them here. In the following experiments, we examine a simulated network on a field of 300 by 300 distance units. Total number of nodes is 3600, corresponding to a density of 0.04. Node sensing radius is 20 units; communication radius is 10. Sensors generate spurious messages due to noise with probability 1% in each epoch. Sending a packet incurs unit energy cost per hop, and the initial battery capacity is arbitrarily chosen to be 1000. We examined, but do not present results from, simulations with the following parameters: density={0.02, 0.03, 0.04, 0.05, 0.06}, noise level={0.01,0.05,0.10}, initial battery power={1000, 2000}. The choice of a particular density, noise level, or initial battery power does not materially impact our results. The choice of epoch duration is arbitrary and application-dependent from the point of view of the system and should be made in consultation with applications. Each epoch contains at least one event in our simulations. An event may span up to 10 epochs and move through the field with a uniformly chosen velocity between 0.0 and 2.0 distance units/epoch. We assume that nodes are stationary after the initial deployment, though none of the nodes make any assumptions about geographical location of other nodes. Every data point represents an average of five runs.

4.5 Results and Discussion

Figure 4 illustrates the impact of our algorithms on system longevity. In this simulation, we have arbitrarily defined system failure as the point when half of the field is no longer being sensed, that is, only half of the field area is within the sensing radius of at least one live sensor which can communicate along some functioning route with the home node. *NetPull* and *NetCenter* lengthen the operational lifetime of the system by a factor of four to five.

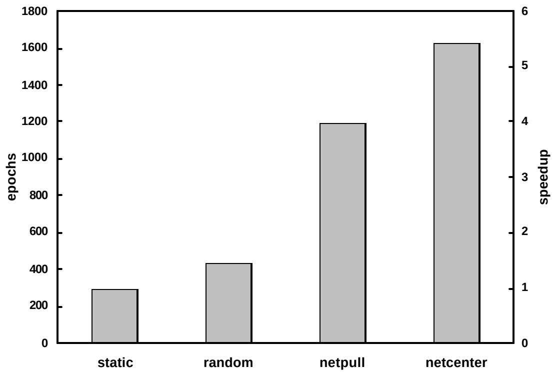

Figure 4: System longevity improvement

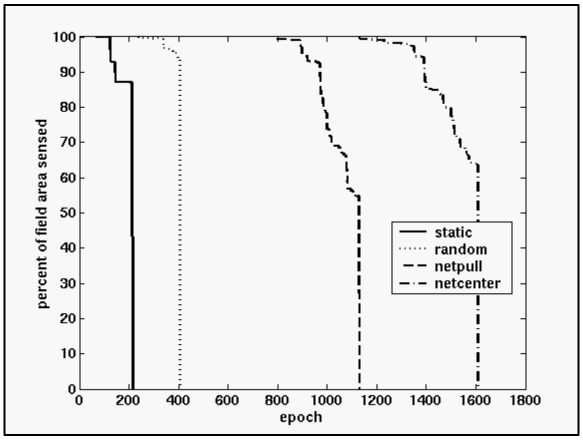

Figure 5: Sensor coverage degradation over time

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

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

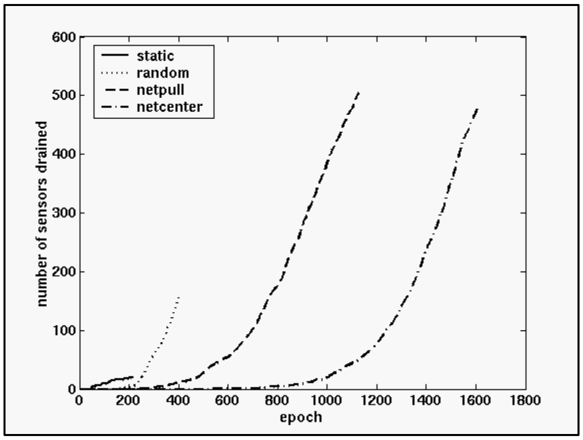

Figure 6: Sensor drainage over time

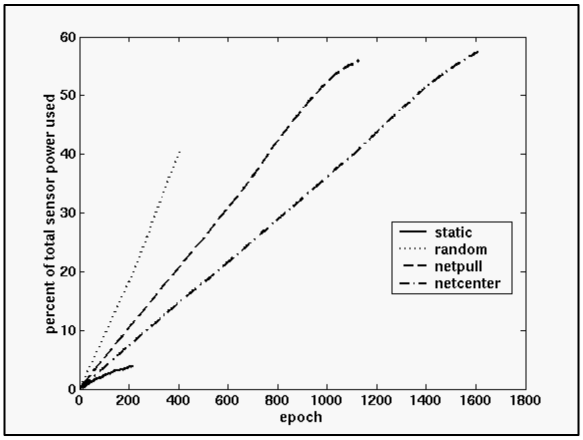

Figure 7: Field energy used over time

Figure 7 shows that *NetPull* and *NetCenter* lie between *Static* and *Random* in terms of energy consumption per epoch. The low slope for *Static* shows that this algorithm uses less energy per epoch because it does not expend any power on active object migration. Most of this power usage is concentrated in a ring around critical nodes, however, and the application terminates early, leaving more than 95% of the total energy on the field unutilized. *NetPull* and *NetCenter* consume more energy than

Static, because they need to move components, but consume less energy than *Random*, because they take application behavior into account. Overall, they outlast both random and static despite lying between them.

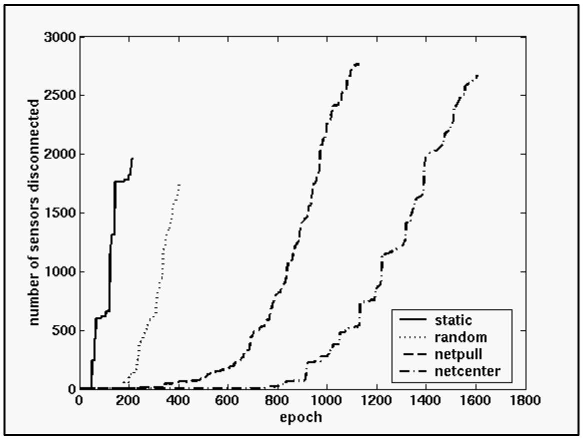

Figure 8: Sensors disconnected over time

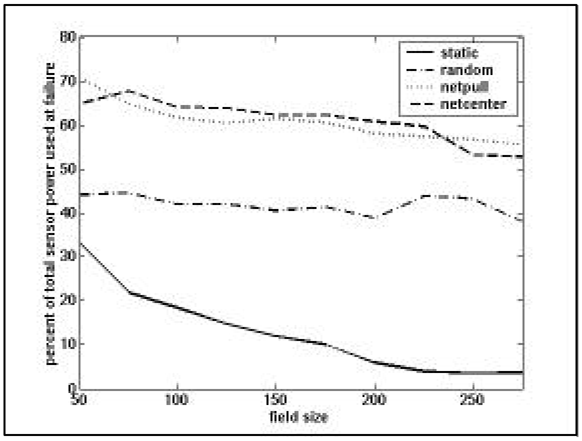

Figure 9: Energy used at breakdown vs field size

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

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

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

5 Issues

In order to gain acceptance, any automatic scheme for code migration should be at least as good as any manual scheme for migrating components. To ensure that our system is no worse than any manual scheme, we provide an explicit interface by which application writers can identify code components, including both class and object instances, and establish affinities between components and ad-hoc nodes. We provide two levels of affinity. Specifying a “strong” affinity between a component and a node effectively anchors the code to that node. This is intended for attaching components like device drivers to the nodes with the installed device in them. Specifying a “weak” affinity immediately migrates the component to the named node, and allows the automated code placement techniques described herein to adapt to the application’s communication pattern from the new starting point. Note that today’s manually constructed applications correspond to the use of strong affinity in our system – unless explicitly moved, components are bound to nodes. The result of overusing strong affinity is a fragile system, where unforeseen communication and mobility patterns can leave an application stranded. While we provide these primitives, we do not advocate their use and believe that automated techniques can outperform manual efforts to place components.

A second issue with deploying Java virtual machines has to do with the ability of cheap, small sensors to support the requisite services required by a Java VM. Java virtual machines on the desktop can indeed have excessive resource requirements, necessitating a fast processor for just-in-time compilation, RAM for expanded object storage, and persistent storage for the system libraries. Previous work [Sirer et al. 99] has directly addressed these issues and proposed a new system architecture for virtual machines that can drastically reduce these resource requirements. Indeed, there are Java Card virtual machines in existence today that are based on a similar, partitioned service architecture that fit on a flexible credit card and cost a few dollars. We assume that running a Java interpreter, or equivalent functionality, on sensors is a solvable problem and concentrate on the distributed coordination of applications.

6 Future Work

In the near future, we plan to extend this work in three directions. First, we have focused on highly dynamic versions of *NetCenter* and *NetPull*, which aggressively move components as soon as they identify an opportunity to save power. In some applications, where the communication patterns form a time-varying graph, such an aggressive approach can lead to inefficient, cyclic behavior. We intend to study variants of these algorithms with differing amounts of hysteresis to dampen feedback loops. Second, we plan to extend our simulations to include node mobility. Our current results indicate that *NetCenter* and *NetPull* will adapt well to changes in network topology. We plan to add node mobility to our simulator and examine the results firsthand. Finally, we plan to investigate the use of our single system image Java operating system with more diverse access patterns and memory allocation. We currently perform local garbage collection on each node, but do not perform distributed garbage collection. We intend to integrate a standard distributed garbage collection algorithm into our system.

7 Conclusion

In this paper, we present the design of a single system image operating system for ad-hoc networks. Our system implements the Java Virtual Machine interface on top of a collection of ad-hoc nodes. An application partitioning tool takes monolithic Java application and converts them into distributed, componentized applications. A small runtime on each node is responsible for object creation, invocation and migration. We rely on a transparent RPC for node-independent communication between components.

Two algorithms, *NetPull* and *NetCenter*, for automatically migrating application components in the network form the core of our operating system. These algorithms are practical, entail low overhead and are easy to implement because they rely only on local information that is easily available from the network. We have demonstrated that they can achieve a factor of four to five improvement in system longevity.

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

8 References

[Amiri 00]	Khalil Amiri, David Petrou, Greg Ganager and Garth Gibson. Dynamic Function Placement in Active Storage Clusters. In *Proceedings of the 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.
[Awerbuch 87]	Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, 230-240, 1987.
[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.
[Broch et al. 98a]	J. Broch, D. A. Maltz, D. B. Johnson, Y. C. Hu, and J. Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad-Hoc Network Routing Protocols. In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, October 1998.

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

[Brunekreef et al. 96] J. Brunekreef, J.-P. Katoen, R. Koymans and S. Mauw, Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9 (4) 157-171, 1996

[Chang and Roberts 79] E.G. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configuration of processors. Communications of the ACM. 22 (5) 281-3, 1979.

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

[Garcia-Molina 82] Hector Garcia-Molina. Elections in Distributed Computer Systems. IEEE Transactions on Computers. C-31 (1) 48-59, 1982.

[Gehani 97] Samir B. Gehani. xDU: A Java-based Framework for Explicitly Partitioning Applications into Distributable Units. Master of Science Thesis, Department of Computer Science, University of San Francisco, 1997

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

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

[Haas & Pearlman 98] Z. J. Haas and M. R. Pearlman, The zone routing protocol (ZRP) for ad-hoc networks (Internet-Draft). Mobile Ad-hoc Network (MANET) Working Group, IETF, Aug. 1998.

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

[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.

[Jacquet et al. 98] P. Jacquet, P. Muhlethaler and A. Qayyum. Optimized Link State Routing Protocol. Internet Draft, draftietf -manet-olsr-00.txt, 1998.

[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 Symposium on Operating System Principles*, December 1995.

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

[Ko & Vaidya 00] Y.-B. Ko and N. H. Vaidya, GeoTORA: A protocol for geocasting in mobile ad-hoc networks. Tech. Rep. 00-010, Dept. of Computer Science, Texas A&M University, March 2000.

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

[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 Computer Sciences Technical Report #1346, University of Wisconsin-Madison, April 1997.

[MANET 01] IETF Mobile Ad-hoc Networking Working Group. Mobile Ad-hoc Networking (MANet). http://tonnant.itd.nrl.navy.mil/manet/manet_home.html.

[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.

[Malpani et al. 00] Navneet Malpani, Jennifer L. Welch and Nitin Vaidya. Leader Election Algorithms for Mobile Ad Hoc Networks. In *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*. 96--103, August 2000.

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

[McCanne & Floyd 01] Steven McCanne and Sally Floyd, UCB/LBNL/VINT Network Simulator - ns (version 2), http://wwwmash. CS.Berkeley.EDU/ns/.

[Ostrovsky 94] Rafail Ostrovsky, Sridhar Rajagopalan and Umesh Vazirani. Simple and efficient leader election in the full information model. In *Proceedings of the 26th annual ACM Symposium on Theory of Computing*, 234-42, May 1994

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

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

[Pei et al. 00] G. Pei, M. Gerla and X. Hong. LANMAR: Landmark Routing for Large Scale Wireless Ad-hoc Networks with Group Mobility. In *Proceedings of IEEE/ACM MobiHOC 2000*, Boston, MA, Aug. 2000.

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

[Plank et al. 95] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the 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.

[Russell et al. 99] Alexander Russell, Michael Saks and David Zuckerman. Lower bounds for leader election and collective coin-flipping in the perfect information model. In *Proceedings of the 31st ACM Symposium on Theory of Computing*. 339-47, May 1999.

[Satyanarayanan 96] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*. Philadelphia, PA, May 1996.

[Sayeed et al. 95] H.M. Sayeed, M. Abu-Amara and H. Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links. *Distributed Computing*, 9 (3) 147-156, 1995.

[Singh 96] G. Singh, Leader election in the presence of link failures. *IEEE Transactions on Parallel and Distributed Systems*, 7 (3). 231-236, March 1996.

[Sirer et al. 99] Emin Gun Sirer, Robert Grimm, Arthur J. Gregory and Brian N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Proceedings of the Seventeenth Symposium on Operating Systems Principles*, pages 202-216, Kiawah Island, South Carolina, December 1999.

[Steketee 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. In *Proceedings of the International Parallel Processing Symposium*, pp. 526--531, Honolulu, HI, April 1996.

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

[Weiser et al. 94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pp. 13-23, Monterey, California, November 1994.

[Wu & Tay 99] C.W. Wu and Y.C. Tay. AMRIS: A Multicast Protocol for Ad-hoc Wireless Networks. In *Proceedings of the IEEE Military Communications Conference* (MILCOM), Atlantic City, NJ, November 1999, pp. 25-29.

[Zayas 87] E. Zayas. Breaking the Process Migration Bottleneck. In *Proceedings of the Eleventh Symposium on Operating SystemsPrinciples*, pp. 13--22, Austin, TX, November 1987.