USENIX Association

Proceedings of the First Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA
March 29–31, 2004

© 2004 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Programming Sensor Networks Using Abstract Regions

Matt Welsh and Geoff Mainland
Harvard University
{mdw,mainland}@eecs.harvard.edu

Abstract

Wireless sensor networks are attracting increased interest for a wide range of applications, such as environmental monitoring and vehicle tracking. However, developing sensor network applications is notoriously difficult, due to extreme resource limitations of nodes, the unreliability of radio communication, and the necessity of low power operation. Our goal is to simplify application design by providing a set of programming primitives for sensor networks that abstract the details of low-level communication, data sharing, and collective operations.

We present *abstract regions*, a family of spatial operators that capture local communication within regions of the network, which may be defined in terms of radio connectivity, geographic location, or other properties of nodes. Regions provide interfaces for identifying neighboring nodes, sharing data among neighbors, and performing efficient reductions on shared variables. In addition, abstract regions expose the tradeoff between the accuracy and resource usage of communication operations. Applications can adapt to changing network conditions by tuning the energy and bandwidth usage of the underlying communication substrate. We present the implementation of abstract regions in the TinyOS programming environment, as well as results demonstrating their use for building adaptive sensor network applications.

1 Introduction

Sensor networks are an emerging computing platform consisting of large numbers of small, low-powered, wireless "motes" each with limited computation, sensing, and communication abilities. Sensor networks are being investigated for applications such as environmental monitoring [8, 27], seismic analysis of structures [7, 20], and tracking moving vehicles [29]. Still, sensor network programming is incredibly difficult, due to the limited capabilities and energy resources of each node as well as the unreliability of the radio channel.

As a result, application designers must make many complex, low-level choices, and build up a great deal of machinery to perform routing, time synchronization, node localization, and data aggregation. To date, little of this machinery has carried over directly from one application to the next, as it encapsulates application-specific tradeoffs in terms of complexity, resource usage, and communication patterns. In this paper, we investigate a suite of general-purpose communication primitives for sensor networks that provide addressing, data sharing, and reduction within local regions of the network. These primitives, which we call *abstract regions*, expose control over the resource consumption of communication, and provide feedback to applications on the accuracy and completeness of collective operations.

A key goal of sensor network programming is to save energy and increase the lifetime of the system by trading increased computation for reduced radio communication (which is relatively expensive in terms of energy cost [17]). Rather than collect samples centrally, it is generally desirable to perform local compression, aggregation, or summarization within the sensor network to reduce overall communication overheads. As a simple example, consider determining the boundary of a region of interest in the network. While this is straightforward to implement centrally if all sensor data is known, a localized version would operate by exchanging data between nearby sensors. Communicating only the boundary threshold to the base station yields significant communication and energy savings.

The core difficulty with building sensor network applications is that high-level, global behavior must be expressed in terms of complex, local actions taken at each node. In the boundary finding example, instead of running standard edge-detection algorithms on the complete set of sensor data, it is necessary to implement techniques that rely chiefly on local data and limited communication between nearby nodes. This challenge is amplified by the limitations of sensor hardware, making traditional parallel algorithms unattractive: such decomposition generally assumes the existence of efficient, any-to-any communication between processors.

In contrast, sensor applications focus on local communication within the network, allowing nodes to coordinate activity with few radio messages. For example, if nodes are able to perform boundary detection by communicating only with other nodes within single-hop (broadcast) radio range, no routing is required, simplifying protocol design and saving energy. This form of local, spatial processing is a powerful paradigm for in-network processing, and is found in a number of existing applications. Currently, developers are required to build up their own mechanisms for such local coordination. Our goal is to simplify application design by identifying a set of reusable, general-purpose communication

primitives based on local regions.

Given such extreme resource limitations, it is critical that any communication abstraction for sensor networks allow the application some measure of control over resource consumption. Unlike more traditional environments, where the primary goal is performance, managing communication overheads is essential for meeting energy and bandwidth goals. Moreover, given the reactive nature of sensor networks, there is the opportunity to design applications that can adapt to changing network or environmental conditions, tuning the bandwidth or energy usage of the communication model to achieve given targets of latency, accuracy, or lifetime.

In this paper, we describe *abstract regions*, a family of spatial operators that capture local communication within regions of the network, which may be defined in terms of radio connectivity, geographic location, or other properties of nodes. In addition to providing a flexible means of node addressing, abstract regions support data sharing using a tuple-space-like programming model, as well as efficient aggregation operations. In addition, abstract regions expose the tradeoff between the accuracy and resource usage of communication operations. Applications can adapt to changing network conditions by tuning the energy and bandwidth usage of the underlying communication substrate. Abstract regions provide feedback on the accuracy of collective operations as well as an interface for controlling resource usage. An earlier position paper [33] introduced the abstract regions interface and its use for tuning energy/accuracy tradeoffs. In this paper, we expand on these ideas and present a detailed study of their applications and performance.

Abstract regions are general enough to support a wide range of sensor network applications and form the basis for other, higher-level communication models. In this paper, we describe the abstract regions programming model, as well as several implementations, including regions based on geographic and radio neighborhoods, spanning trees, and an approximate planar mesh. We have implemented abstract regions in nesC [11], a component-oriented variant of C used by the TinyOS [17] operating system. To further simplify abstract region programming, we have implemented a lightweight, thread-like abstraction for TinyOS, allowing applications to invoke blocking operations, which were heretofore disallowed by the TinyOS concurrency model.

We present four applications based on the abstract region model: tracking a moving object through a sensor field, detecting a contour in sensor readings, event detection based on a variant of directed diffusion [18], and geographic routing using GPSR [19]. We evaluate the accuracy and communication overhead of each of these applications and present results highlighting the tradeoff between resource consumption and quality of the results. The rest of this paper is organized as follows. Section 2 motivates our approach and discusses related work. Section 3 introduces the abstract region programming model and the use of resource tuning. Section 4 describes the implementation of abstract regions in the TinyOS operating system. Section 5 presents three applications based on regions, and Section 6 evaluates the accuracy and resource consumption tradeoff provided by the regions programming model. Section 7 discusses future work and concludes.

2 Motivation and Background

Sensor networks have attracted increasing interest from research and industry. The potential to instrument the physical world at high resolution and low cost opens up a wide range of novel applications in areas such as engineering [7, 20], biology [8, 27], and medicine [9, 34]. The future success of sensor networks depends to a large extent on the programming and communication abstractions presented to application developers. This is especially critical when one considers that the primary developers for sensor networks will be scientists and engineers. These users require a programming model that closely represents the high-level, data-centric computation to be performed within the network, rather than the existing, low-level, node-centric programming interfaces provided by existing systems.

In this paper, we focus on sensor networks based on small, low-power sensors such as the UC Berkeley MICA node. This device consists of a 7.3 MHz ATmega128L processor, 128KB of code memory, 4KB of data memory, and a Chipcon CC1000 radio capable of 38.4 Kbps and an outdoor transmission range of approximately 300m. The device measures 5.7cm $\times$ 3.1cm $\times$ 1.8cm and is typically powered by 2 AA batteries with an expected lifetime of days to months, depending on application duty cycle. The limited memory and computational resources of this platform make an interesting design point, as software layers must be tailored for this restrictive environment. The MICA node uses a lean, component-oriented operating system, called TinyOS [17], and an unreliable message-passing communication model based on Active Messages [32].

2.1 The need for abstractions

The bandwidth and energy limitations of sensor nodes typically require that in-network processing be performed to reduce the amount of data that must be transferred out of the network. Application designers are therefore faced with the problem of decomposing an initially straightforward data-collection task into a parallel program with local communication among sensor nodes. Currently, sensor application designers spend a great deal of effort building up low-level machinery for routing, data collection, and energy management. We wish to move away from this state of affairs by providing higher-level programming interfaces that abstract these details, yet provide enough flexibility to implement efficient al-

gorithms. The first step in this endeavor is to identify an appropriate set of programming primitives, and is the primary focus of this paper.

Our approach can be likened to that of MPI [13], which provides a unified interface for message passing across a large family of parallel machines. MPI hides the details of the communication hardware and provides efficient implementations of common collective operations, such as broadcast and reduction. MPI has been extremely successful in the parallel processing community as it is high-level enough to shield programmers from most of the details of the underlying machine, yet low-level enough to permit extensive application-specific optimizations. We wish to provide communication interfaces that serve a similar role for sensor networks.

2.2 Related work

Several communication and programming models for sensor networks have been proposed in the literature. Among the first was directed diffusion [18], which provides a framework for distributed event detection and propagation. This approach has been extended by Heidemann *et al.* [14] to support in-network filtering and aggregation. In contrast, our focus is on a lower-level mechanism, namely, maintaining and coordinating regions of nodes for in-network processing. Later in this paper, we show that abstract regions can be used to implement a form of directed diffusion (see Section 5.3). Other communication abstractions include GHT [30], Spatial Programming [3], DIFS [12], SPIN [15], and DIMENSIONS [10]. These systems are generally focused on a specific communication or aggregation model rather than supporting a wide range of applications.

Most closely related to our approach is Hood [35], a neighborhood-based programming abstraction for sensor networks. Like abstract regions, Hood provides a mechanism for discovery and sharing of data among sensor nodes. The shared variable mechanism described in Section 3 was inspired by the Hood model, although Hood provides a richer set of primitives for manipulating shared data. Unlike abstract regions, Hood does not directly address the resource/accuracy tradeoff or provide a mechanism for data aggregation.

TinyDB [25], Cougar [38], and IrisNet [28] provide a high-level SQL or XML-based query interface to sensor network data. Queries are deployed into the network, streaming results to one or more base stations, and aggregation is used to reduce communication overhead. These systems have tremendous value and abstract away many details of communication, aggregation, and filtering. By the same token, however, they are not well-suited for developers who wish to implement specific behavior at a lower level than the query interface.

For example, TinyDB is focused on relaying aggregate data along a spanning tree rooted at a base station. While this mechanism can support complex algorithms such as contour finding, TinyDB must expose an internal "contour finding operator" to queries [16]. Our concern is with the implementation of these lower-level operators themselves, and with providing a set of communication abstractions that can be used to implement higher-level services, such as queries.

2.3 Resource management and adaptation

It is critical that communication abstractions for sensor networks take resource management into account, and give applications control over the tradeoff between resource usage and the accuracy and yield of collective operations [33]. The energy usage of communication is dictated by a number of factors, including hardware properties, node density, radio channel quality, and local activity within the network. Moreover, these factors are generally not known *a priori* and may be highly dynamic. As a result, it is often desirable to consume fewer resources to obtain an approximate result, rather than pay an arbitrary resource cost for complete accuracy.

Adlakha *et al.* [1] describe a design-time recipe for tuning aspects of sensor networks to achieve given accuracy, latency, or lifetime goals. However, this approach assumes a statically-configured network where resource requirements are known in advance, rather than allowing the network behavior to be tuned at runtime (say, in response to increased activity).

Several adaptive communication mechanisms for sensor networks have been proposed. SPIN [15] is a set of data dissemination protocols that adapt to energy availability by reducing protocol overhead when energy resources are low. TinyDB provides a *lifetime* keyword that scales the query sampling and transmission period to meet a user-supplied network lifetime [26]. Boulis *et al.* [4] describe an aggregation mechanism that exposes an energy/accuracy knob to the user. These approaches are a step in the right direction, and our goal is to generalize the communication abstraction exposed to applications while retaining this approach of yielding control over resource usage.

3 Abstract Regions

Sensor network applications are often expressed in terms of groups of nodes over which local sampling, computation, and communication occur. For example, tracking a moving object involves aggregating sensor readings from nodes near the object. Abstract regions are a communication abstraction intended to simplify application development by providing a region-based collective communication interface. Abstract regions capture the inherent locality of communication and hide the details of data dissemination and aggregation within regions.

An abstract region defines a *neighborhood relationship* between a particular node and other nodes in the network. Examples of neighborhood predicates include "the set of nodes within N radio hops" and "the set of

nodes within distance d." Local, spatial operations in the sensor network are performed by sharing data and coordinating activity among nodes in this neighborhood. In general, each node in the sensor network defines multiple abstract regions that it wishes to operate over, depending on application requirements. A node may also be a member of multiple regions at once: for example, a node will belong to multiple single-hop radio regions, one for each of its own set of single-hop neighbors.

3.1 Programming Abstraction

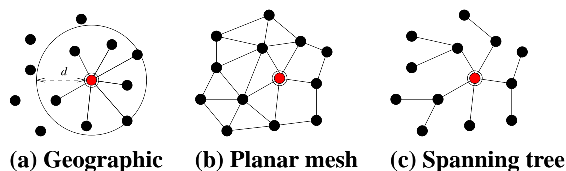
Regions capture a range of common idioms in sensor network programming. These include identification of neighboring nodes, data sharing, and data reduction within local neighborhoods. These operators allow nodes to query the state of neighboring nodes and implement efficient aggregation, compression, and summarization of local data. This programming model is based on that originally proposed by Hood [35]. Abstract regions support the following set of operators:

Neighbor discovery: Before performing other operations on a region, each node initiates the process of discovering neighboring nodes. Depending on the type of region, this may require broadcasting messages, collecting information on node locations, or estimating radio link quality. This is a continuous process, and each node is informed of changes in the region membership set, due to nodes joining, leaving, or moving within the network. A node may terminate this process at any time to avoid additional discovery messages. When terminated, the neighbor discovery operator returns a quality metric that measures the accuracy of the region formation, such as the percentage of candidate nodes that responded to the discovery request.

Enumeration: The enumeration operator returns the set of nodes participating in the region, allowing them to be addressed, for example, for direct message communication. Supplemental information, such as the location of each node, may be returned as well.

Data sharing: The data sharing operator allows variables, represented as key/value pairs, to be shared among nodes in the region. A shared variable supports two operations: *get* and *put*. *get(v,n)* retrieves the value of variable v from node n, and *put(v,l)* stores the value l of variable v at the local node. A simple implementation of *get(v,n)* sends a message to node n requesting the value of v. In this case, *put(v,l)* simply stores the value of v locally. An alternate implementation might broadcast or gossip the values of shared variables among nodes in the region, allowing *get(v,n)* to fetch locally cached values.

Reduction: The reduction operator takes a shared variable key and an associative operator (such as *sum*, *max*, or *min*) and reduces the shared variable across nodes in the region, storing the result in a shared variable. Reduction may be implemented in a number of ways, such as by collecting all values locally, or forming a spanning tree and propagating values up the branches of the tree, performing reduction at each level. As with shared variables, the region hides the details of reductions from the programmer.



(a) Geographic **(b) Planar mesh** **(c) Spanning tree**

Figure 1: **Examples of abstract regions.**

Abstract regions simplify application design by shielding developers from the complexity of routing, data dissemination, and state management. Additionally, regions provide a unified interface regardless of the particular definition of the region membership. That is, one can readily interchange the underlying region implementation without necessarily affecting higher-level application logic. For example, an application that makes use of an N-radio-hop region can be readily modified to use a geographic neighborhood region in its place.

3.2 Abstract Region Implementations

Given the diverse needs of sensor network applications, we expect a range of abstract region definitions will be useful to programmers. We have completed several abstract region implementations, with several others underway. Three examples are shown in Figure 1. They include:

- *N-radio hop:* Nodes within N radio hops;
- *N-radio hop with geographic filter:* Nodes within N radio hops and distance d;
- *k-nearest neighbor:* k nearest nodes within N radio hops;
- *k-best neighbor:* k nodes within N radio hops with the highest link quality, as measured in fraction of packets dropped over some measurement interval;
- *Approximate planar mesh:* A mesh with a small number (possibly zero) crossing edges; and
- *Spanning tree:* A spanning tree rooted at a single node, used for aggregating values over the entire network.

3.2.1 Radio and geographic neighborhoods

The implementation of the radio and geographic neighborhoods is straightforward. To discover neighbors, each node broadcasts periodic advertisements. Data sharing may be implemented using either a "push" (*put* broadcasts updates to neighboring nodes) or "pull" (*get* sends a fetch message to the corresponding node) approach;

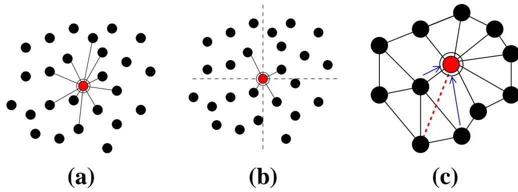


Figure 2: **Forming an approximate planar mesh region.** *(a) Nodes first create a region of k-nearest neighbors. (b) The nearest neighbor in each sector is chosen as an outedge and advertised. (c) If an advertised edge crosses another edge, invalidation messages are sent to the source.*

our current implementation uses the latter. Reduction involves collecting shared variable values locally, combining them with the reduction operator, and storing the result in another shared variable.

3.2.2 Approximate planar mesh

Planar meshes, such as the Delaunay triangulation [31], are useful for spatial computation (such as dividing space into nonoverlapping cells) and geographic-based routing [19]. However, algorithms for constructing planar meshes generally require either global knowledge of the connectivity graph or extensive interprocessor communication. We have implemented an *approximate* planar mesh in which a small number of edges may cross, but which uses communication only within single-hop neighborhoods of each node.

Our algorithm is based on a pruned Yao graph [23] and is depicted in Figure 2. First, each node discovers a k-nearest radio region of candidate nodes. When k neighbors have been accumulated, each node forms the Yao graph by dividing space around it into m equal-sized sectors of angle $\theta = 2\pi/m$ and selecting the nearest node within each sector as a potential neighbor. Next, each node uses a single-hop broadcast to advertise its selected outedges. Upon reception of an edge advertisement, each node tests whether the given edge crosses one of its own outedges, and if so sends an invalidation message to the source, causing it to prune the offending edge from its set. Nodes do not select additional neighbors beyond the initial candidates, so a node may end up with fewer than m outedges. Nodes perform several rounds of edge set broadcasts and wait for some time before settling on a final set of neighbors. Both the number of broadcasts and the timeout settings can be tuned by the application, as discussed below in Section 3.3. Changes in the underlying k-nearest-neighbor set initiates selection of new Yao neighbors. Data sharing and reduction are implemented using the same components as in the radio neighborhood, as all neighbors are one radio hop away.

We have also implemented planar regions based on the Relative Neighborhood Graph and Gabriel Graph, and an implementation of GPSR [19] based on them. This is described in Section 5.4.

3.2.3 Spanning tree

Spanning trees are useful for aggregating values within the sensor network at a single point, as demonstrated by systems such as TinyDB [25] and directed diffusion [18]. Our spanning tree implementation provides the same programmatic interface as other regions, although the semantics of the shared variable and reduction operations have been augmented to support *global* communication through the routing topology provided by the spanning tree. The shared variable *put* operation at the root floods the shared value to all nodes in the tree, while a *put* at a non-root node propagates the value to the root. Reduction operations cause data to propagate up the tree, causing each node to aggregate its local value with that of its children.

The spanning tree implementation adapts to changing network conditions, as nodes attempt to select a parent in the spanning tree to maximize the link quality and minimize the message hopcount to the root. Each node maintains an estimate of the link quality to its parent, measured in terms of the fraction of successful message transmissions to the parent and the fraction of messages successfully received from other nodes, using a simple sequence number counting scheme.

The tree is formed by broadcasting messages indicating the source node ID and number of hops from the root; nodes rebroadcast received advertisements with their own ID as the source and the hopcount incremented by 1. Nodes initially select a parent in the tree based on the lowest hopcount advertisement they receive, but may select another parent if the link quality estimate falls below a certain threshold. This estimate is smoothed with an exponentially weighted moving average (EWMA) to avoid rapid parent reselections. Our goal in this work has not been to develop the most robust tree formation algorithm, but rather to demonstrate that spanning trees fit within the programming interface provided by abstract regions. It would be straightforward to replace the tree construction algorithm with another, such as that described in [36], and layer the abstract regions interface over it.

3.3 Quality feedback and tuning interface

The collective operations provided by abstract regions are inherently unreliable. The quality of region discovery, the fraction of nodes contacted during a reduction operation, and the reliability of shared variable operations all depend on the number of messages used to perform those operations. Generally, increasing the number of messages (and hence, energy usage) improves communication reliability and the accuracy of collective operations. However, given a limited energy or bandwidth budget, the application may wish to perform region operations with reduced fidelity.

Abstract regions expose this tradeoff between resource consumption and the accuracy or yield of collec-

tive operations. This is in contrast to most existing approaches to sensor network communication, which hide these resource tradeoffs from the application. As a concrete example of such a tradeoff, the communication layer may tune the number and frequency of message retransmissions to obtain a given degree of reliability from the underlying radio channel. Similarly, nodes may vary the rate at which they broadcast location advertisements to neighboring nodes, affecting the quality of region discovery.

Abstract regions provide feedback to the application in the form of a *quality measure* that represents the completeness or accuracy of a given operation. For member discovery, the quality measure represents the fraction of candidate nodes that responded to the discovery request. For example, when discovering one-hop neighbors within distance d, the quality measure represents the fraction of one-hop nodes that responded to the region formation request. For reduction, the quality measure represents the fraction of nodes in the region that participated in the reduction. In addition, each operation supports a timeout mechanism, in which the operation will fail if it has not completed within a given time interval. For example, when performing a shared variable *get* operation, a timeout indicates that the data could not be retrieved from the requested node.

Applications can use this quality feedback to affect resource consumption of collective operations through a *tuning interface*. The tuning interface allows the application to specify low-level parameters of the region implementation, such as the number of message broadcasts, amount of time, or number of candidate nodes to consider when forming a region. Likewise, region operations perform message retransmission and acknowledgment to increase the reliability of communication; the depth of the transmit queue and number of retransmission attempts can be tuned by the application.

In our current implementation, the set of parameters exposed by the tuning interface is somewhat low-level and in many cases specific to the particular region implementation. Many of these parameters are straightforward to tune, although their effect on resource consumption may not be immediately evident, as it is highly dependent on application behavior. In general, an application designer will need to study the impact of the relevant tuning parameters on resource usage and application performance.

Exposing tuning knobs and quality feedback greatly impacts the programming model supported by abstract regions. Rather than hiding performance tradeoffs within a generic communication library with no knowledge of application requirements, the tuning interface enables applications to adapt to changing environments, network conditions, node failure, and energy budget. For example, this interface can be used to implement adaptive control mechanisms that automatically tune region parameters to meet some quality or resource usage target. In Section 6, we demonstrate an adaptive controller that tunes the maximum retransmission count to achieve a target yield for reduction operations.

We are currently working on a resource-centric tuning mechanism that allows the programmer to express an energy, radio bandwidth, or latency budget for each operator, and which maps these constraints onto the appropriate low-level parameter settings. This approach shields the programmer from details of the low-level interface and enables adaptivity using higher-level resource constraints. This is discussed further in Section 7.

4 Implementation

In this section we detail the implementation and programming interface for abstract regions on TinyOS [17], a small operating system for sensor nodes. Unlike the event-driven, asynchronous concurrency model provided by TinyOS, abstract regions provide a blocking, synchronous interface, which greatly simplifies application logic. This is accomplished using a lightweight, thread-like abstraction called *fibers*, described below.

4.1 TinyOS concurrency model

Traditional concurrency mechanisms, such as threads, are too heavyweight to implement on extremely resource-constrained devices such as motes. TinyOS employs an event-driven concurrency model, in which long-running operations are not permitted to block the application, but rather using a *split-phase* approach. An initial request for service is invoked through a *command*, and when the operation is complete, an *event* (or callback) is invoked on the original requesting component. For example, to take a sensor reading, an application invokes the `getData()` command on the sensor hardware component, which later invokes the `dataReady()` callback supplying the sensor reading.

The TinyOS concurrency model requires each concurrent "execution context" to be implemented manually by the programmer as a state machine, with execution driven by the sequence of commands and events invoked on each software component. If an application is performing multiple concurrent tasks, these operations must be carefully interleaved. In addition, each split-phase operation requires that the application code be broken across multiple disjoint segments of code. The programmer must manually maintain continuation state across each split-phase operation, adding significant complexity to the code. While the logical program may be quite simple, the lack of blocking operations in TinyOS requires that the application be broken into multiple tasks and event handlers.

A key observation is that a broad class of sensor network applications can be represented by a small number of concurrent activities: for example, a core application

loop that periodically performs some sensing and communication operations, as well as a reactive context that responds to external events (e.g., incoming messages). Providing blocking operations in TinyOS does not require a full-fledged threads mechanism, but rather the means to support a limited range of blocking operations along with event-driven code to handle asynchronous events.

4.2 Lightweight Fibers

We have implemented a lightweight, thread-like concurrency model for TinyOS, supporting a single blocking execution context alongside the default, event-driven context provided by TinyOS. We use the term *fibers* to refer to each execution context.[1] The default *system fiber* is event-driven and may not block, while the *application fiber* is permitted to block. Because there is only one blocking context, both the system and application fibers can share a single stack. The overhead for each fiber, therefore, consists of a saved register set, which on the ATmega128L is 24 bytes. The runtime overhead for a context switch is just 150 instructions. Apart from the single blocking context, applications may employ additional concurrency through TinyOS's event-driven concurrency model.

The application fiber may block by saving its register set and jumping to the system fiber. The system fiber restores its registers, but resumes execution on the application stack. An event, such as a timer interrupt or message arrival, may wake up the application fiber, causing its register state to be restored once the event handler has completed. The system fiber maintains no live state on the stack after the application fiber is resumed.

Fibers allow blocking and event-driven concurrency to be freely mixed in an application, and different components can use different concurrency models depending on their requirements. In general, the central sense-process-communication loop of an application may employ blocking for simplicity, while the bulk of the TinyOS code remains event-driven.

4.3 Abstract region API

The abstract regions programming interface is shown in Figure 3. The use of blocking interfaces, provided by fibers, greatly simplifies application design. Rather than breaking application logic across a set of disparate event handlers, a single, straight-line loop can be written. Apart from considerably shortening the code, this approach avoids common programming errors, such as maintaining continuation state incorrectly.

Without blocking calls, the object tracking application described in Section 5.1 is 369 lines of nesC code, consisting of 5 event handlers and 11 continuation functions. The corresponding blocking version (shown in pseudocode form in Section 5.1) is 134 lines of code and consists of a single main loop. All application state is stored in local variables within the loop, rather than being marshaled into and out of global continuations for each split-phase call.

[1] This term is borrowed from Windows, in which fibers are explicitly scheduled by the application [2]

```
/* Discover region */
result_t Region.formRegion(<region specific args>,
  int timeout);

/* Wait for region discovery */
result_t Region.sync(int timeout);

/* Set local shared variable */
result_t SharedVar.put(sv_key_t key, sv_value_t val);

/* Get shared variable from give node */
result_t SharedVar.get(sv_key_t key, addr_t node,
  sv_value_t *val, int timeout);

/* Wait for shared variable gets */
result_t SharedVar.sync(int timeout);

/* Reduce 'value' to 'result' with given op */
/* 'yield' returns pct of nodes responding */
result_t Reduce.reduceToOne(op_t operator,
  sv_key_t value, sv_key_t result,
  float *yield, int timeout);

/* Reduce and set result in all nodes */
result_t Reduce.reduceToAll(op_t operator,
  sv_key_t value, sv_key_t result,
  float *yield, int timeout);

/* Wait for reductions to complete */
result_t Reduce.sync(int timeout);
```

Figure 3: **Abstract regions programming interface.**

Abstract regions themselves are implemented using event-driven concurrency, due to the number of concurrent activities that the region must perform. For example, the spanning tree region is continually updating routing tables based on link quality estimates received from neighboring nodes; the reactive nature of this activity is better suited to an event-driven model. A wrapper component is used to provide a blocking, fiber-based interface to each abstract region implementation, and applications can decide whether to invoke a region through the blocking or the split-phase interface.

5 Applications

To demonstrate the effectiveness of regions, in this section we describe four sensor network applications that are greatly simplified by their use. These include tracking an object in the sensor field, finding spatial contours, distributed event detection using directed diffusion [18], and geographic routing using GPSR [19].

5.1 Object tracking

Object tracking is an oft-cited application for sensor networks [5, 22, 37]. In its simplest form, tracking involves determining the location of a moving object by detecting changes in some relevant sensor readings, such as magnetic field.

Our version of object tracking uses a simple algorithm based on the DARPA NEST demonstration software, described in [35]. Each node takes periodic magnetometer readings and compares them to a threshold value. Nodes above the threshold communicate with their neighbors and elect a leader node, which is the node with the largest magnetometer reading. The leader computes the centroid of its neighbors sensor readings, and transmits the result to a base station.

The following pseudocode shows this application expressed in terms of abstract regions:[2]

```
location = get_location();
/* Get 8 nearest neighbors */
region = k_nearest_region.create(8);

while (true) {
  reading = get_sensor_reading();

  /* Store local data as shared variables */
  region.putvar(reading_key, reading);
  region.putvar(reg_x_key, reading * location.x);
  region.putvar(reg_y_key, reading * location.y);

  if (reading > threshold) {
    /* ID of the node with the max value */
    max_id = region.reduce(OP_MAXID, reading_key);

    /* If I am the leader node ... */
    if (max_id == my_id) {
      /* Perform reductions and compute centroid */
      sum = region.reduce(OP_SUM, reading_key);
      sum_x = region.reduce(OP_SUM, reg_x_key);
      sum_y = region.reduce(OP_SUM, reg_y_key);
      centroid.x = sum_x / sum;
      centroid.y = sum_y / sum;
      send_to_basestation(centroid);
    }
  }
  sleep(periodic_delay);
}
```

The program performs essentially all communication through the abstract regions interface, in this case the k-nearest-neighbor region. Nodes store their local sensor reading and the reading scaled by the x and y dimensions of their location as shared variables. Nodes above the threshold perform a reduction to determine the node with the maximum sensor reading, which is responsible for calculating the centroid of its neighbors' readings. A series of sum-reductions is performed over the shared variables which are used to compute the centroid:

$$
\begin{aligned}
c_x &= \sum_i R_i x_i / \sum_i R_i \\
c_y &= \sum_i R_i y_i / \sum_i R_i
\end{aligned}
$$

where c_x and c_y are the (x, y)-coordinates of the centroid, R_i is the reading at node i, and x_i and y_i are the (x, y)-coordinates of node i.

The use of regions greatly simplifies application design. The programmer need not be concerned with the details of routing, data sharing, or identifying the appropriate set of neighbor nodes for performing reductions. As a result the application code is very concise. As described earlier, our actual implementation is just 134 lines of code, of which 70 lines comprise the main application loop above. The rest consists of variable and constant declarations as well as initialization code.

As a rough approximation of the linecount for a "hand coded" version of this application, without the benefit of the abstract regions interface, consider the combined size of the nonblocking object tracking code (369 lines) and the k-nearest-neighbor region code (247 lines). This total of 616 lines, compared with 134 lines for the abstract region version, suggests that this programming abstraction captures a great deal of the complexity of sensor applications.

[2] For brevity, the use of tuning and quality feedback is not shown in this example.

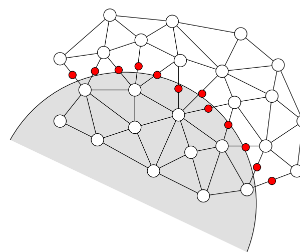


Figure 4: **Contour finding application.** *The shaded region represents an area where sensor readings fall above a threshold. Nodes (unfilled circles) are connected into an approximate planar mesh. Contour points (filled circles) are chosen as midpoints between a node above the threshold and a node below the threshold.*

5.2 Contour finding

Another typical sensor network application is detecting contours or edges in the set of sensor readings [16, 24]. Contour finding involves determining a set of points in space that lie along, or close to, an isoline in the gradient of sensor readings. Contours might represent the boundary of a region of interest, such as a group of sensors with an interesting range of readings. Contour finding is a valuable spatial operation as it compresses the per-node sensor data into a low-dimensional surface. This primitive could be used for detecting thermoclines or tracking the flow of contaminants through soil [6].

An implementation of contour finding using abstract regions is depicted in Figure 4, and is expressed in pseudocode as:

```
location = get_location();
/* Form approximate planar mesh */
region = apmesh_region.create();
region.putvar(loc_key, location);

while (true) {
  reading = get_sensor_reading();
  region.putvar(reading_key, reading);

  if (reading > threshold) {
    foreach (nbr in region.get_neighbors()) {
```

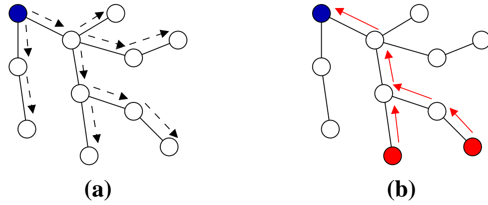


Figure 5: **Directed diffusion operation.** *(a) The sink forms a spanning tree and floods interests throughout the network. (b) Matching nodes route event reports to the corresponding sink through the spanning tree topology.*

```
      /* Fetch neighbor's reading */
      rem_reading = region.getvar(reading_key, nbr);
      if (rem_reading <= threshold) {
        rem_loc = region.getvar(loc_key, nbr);
        contour_point = midpoint(location, rem_loc);
        send_to_basestation(contour_point);
      }
    }
  }
  sleep(periodic_delay);
}
```

Nodes first form an approximate planar mesh region, as described in Section 3. Each node stores its location and sensor reading as a shared variable. Nodes that are above the sensor threshold of interest fetch the readings and locations of their neighbors. For each neighbor that is below the threshold, the node computes a contour point as the midpoint between itself and its neighbor, and sends the result to the base station.

The use of the approximate planar mesh region ensures that few edges will cross, and that nodes will generally select neighbors that are geographically near. These properties are vital for our contour finding algorithm as it is based on pairwise comparisons of sensor readings, and values are computed along edges in the mesh. As with the object tracking application, abstract regions shield the programmer from details of mesh formation and data sharing; the application code is very straightforward. It is only 118 lines of nesC code, 56 lines of which are devoted to the main application loop.

5.3 Directed diffusion

As discussed in Section 2, directed diffusion [18] has been proposed as a mechanism for distributed event detection in sensor networks. The idea is to flood *interests* for named data throughout the network (such as "all nodes with sensor readings greater than threshold T"), as shown in Figure 5. Nodes that match some interest report their local value to the *sink* that generated the interest. Here, we show that abstract regions can be used to implement the directed diffusion mechanism, demonstrating their value for building higher-level communication abstractions.

The directed diffusion programming interface consists of two operations: *broadcastInterest* and *publishData*. Sink nodes invoke *broadcastInterest* with an operator (such as $=$, $>$, or $<=$) and a comparison value. Our current implementation supports interests in the form of simple binary operator comparisons; it would be straightforward to extend this to support more general interest specifications. Nodes call *publishData* to provide local data, which is periodically compared against all received interests. When the local data matches an interest, the node routes an event report to the corresponding sink. When the sink receives an event report, a *dataReceived* event handler is invoked with the corresponding node address and value.

Our directed diffusion layer is implemented using the spanning tree abstract region. Data sinks first form an adaptive spanning tree, and publish an interest record by inserting it into the tuplespace associated with that region. Recall that in the spanning tree region, the tuplespace *put* operation, when invoked at the root, broadcasts the entry to all nodes in the tree. Nodes periodically check for interest records using the tuplespace *get* operation, and compare their current value (provided by `publishData()`) with all received interests. If the value matches, it is inserted into the tuplespace of the corresponding spanning tree, which propagates the value to the root.

Nearly all of the complexity of broadcasting interests and propagating data to the sink is captured by the spanning tree region and its tuplespace implementation. The directed diffusion layer is responsible only for representing interests and matching data values as tuplespace entries, allowing the spanning tree region to handle the details of maintaining routes between data sources and sinks. One drawback with our current implementation is that multiple sinks receiving the same data must form separate spanning trees, causing nodes with matching data to send multiple copies. However, it would be straightforward to optimize the spanning tree region by suppressing duplicate packets being sent along the same network path. The directed diffusion layer is only 188 lines of nesC code, of which 66 lines handle initialization and definitions. In contrast, the spanning tree region code is 937 lines.

5.4 Geographic routing using GPSR

Geographic routing protocols use the locations of nodes in the network to make packet forwarding decisions. Rather than maintaining distance vectors or link state at each node, a node only needs to know the geographic positions of its immediate neighbors. One such protocol is Greedy Perimeter Stateless Routing (GPSR) [19], which operates in two modes: *greedy routing*, in which the message is routed to the neighboring node that is closest to the destination, and *face routing*, which is used when no neighbor is closer to the destination than the current node. Face routing requires a planar mesh to ensure that these local minima can be traversed.

We have implemented GPSR using both our radio

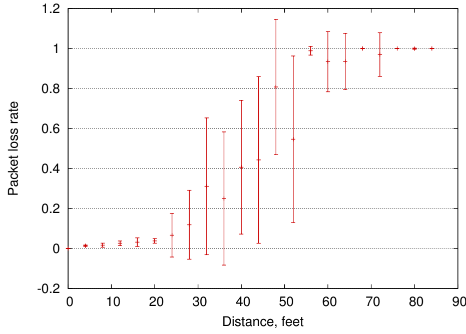


Figure 6: **TOSSIM radio loss model based on empirical data.** *The mean packet loss rate versus distance is shown, with errorbars indicating one standard deviation from the mean. The model is highly variable at intermediate distances.*

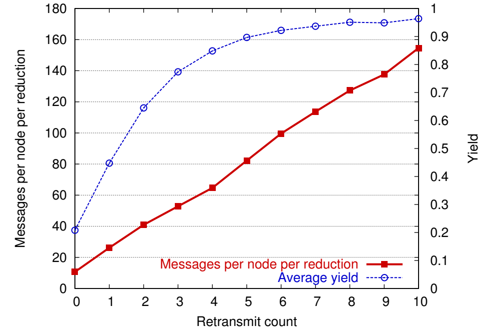


Figure 7: **Reduction yield and overhead in the k-nearest neighborhood region.** *This figure shows the average yield (fraction of neighbors responding to a reduction request) and number of messages for reduction operations. The yield and overhead are directly related to the number of retransmission attempts made by the transport layer.*

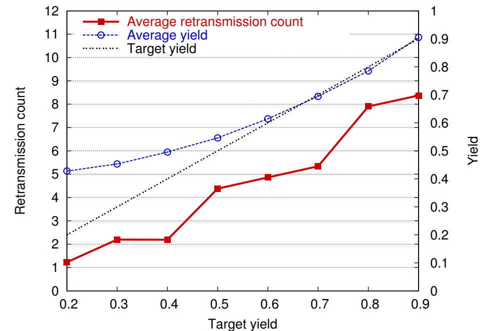


Figure 8: **Adaptive reduction algorithm performance.** *This figure shows the effectiveness of dynamically tuning the maximum retransmission count to meet a reduction yield target. The figure shows the average yield across 100 reduction operations, as well as the average retransmission count determined by the controller.*

neighborhood region (for greedy routing mode) and approximate planar mesh regions (for face routing mode). The abstract regions code handles all aspects of constructing the appropriate graphs, greatly simplifying the GPSR code itself. There are three types of approxiate planar mesh: the pruned Yao graph (PYG), Relative Neighborhood Graph (RNG), and Gabriel Graph (GG). Each of these graphs has slightly different rules for determining inclusion of an edge using only local information about the immediate neighbors of each node. Our GPSR implementation provides a scalable *ad hoc* routing framework for sensor networks, although space limitations prevent us from presenting detailed results.

6 Evaluation

In this section, we evaluate the abstract region primitives and demonstrate their ability to support tuning of resource consumption to achieve targets of energy usage and reliability. We investigate five scenarios: adaptive shared variable reduction, construction of an approximate planar mesh, object tracking, contour finding, and event detection using directed diffusion. In each case, we explore the use of the tuning interface to adjust resource usage and evaluate its effect on the accuracy of region operations.

These results were obtained using TOSSIM [21], a simulation environment that executes TinyOS code directly; our abstract region code can either run directly on real sensor motes or in the TOSSIM environment. TOSSIM incorporates a realistic radio connectivity model based on data obtained from a trace gathered from Berkeley MICA motes in an outdoor setting, as shown in Figure 6. This loss probability captures transmitter interference during the original trace that yielded the model. More detailed measurements would be required to capture the full range of transmission characteristics, although experiments have shown the model to be highly accurate [21]. We simulate a network of 100 nodes distributed semi-irregularly in a 20x20 foot area. Because TOSSIM does not currently simulate the energy consumption of nodes, we report the number of radio messages sent as a rough measure of energy consumption, which is reasonable given that on the MICA platform, radio communication dominates CPU energy usage by several orders of magnitude.

6.1 Adaptive reduction algorithm

By performing reduction over a subset of neighbors in a region, nodes can trade off energy consumption for accuracy. As described earlier, the reduction operator provides quality feedback in the form of the fraction of nodes that responded to the reduce operation. Over an unreliable radio link, this fraction is directly related to the number of retransmission attempts made by the underlying transport layer, as shown in Figure 7. Moreover, the appropriate retransmission count to meet a given yield target is a function of the local network density and channel characteristics, which vary both across the network and over time.

```
region = k_nearest_region.create(8);
region.putvar(key, local_id);
tuning.set(MAX_RETRANSMIT_COUNT, max_xmit);

while (true) {
  val = region.reduce(OP_MAX, key, &quality, timeout);
  avg_quality = (quality * ALPHA) +
    (avg_quality * (1.0 - ALPHA));

  if (avg_quality < (target - LOW_WATER)) {
    max_xmit++;
  } else if (avg_quality > (target + HIGH_WATER)) {
    if (--max_xmit < 1) max_xmit = 1;
  }
  tuning.set(MAX_RETRANSMIT_COUNT, max_xmit);
  sleep(delay);
}
```

Figure 9: **Pseudocode for adaptive reduction algorithm.**

We implemented a simple adaptive reduction algorithm that attempts to maintain a target reduction yield by dynamically adjusting the maximum number of retransmission attempts made by the transport layer. Pseudocode is shown in Figure 9, and illustrates the use of the regions tuning interface. Nodes form a k-nearest-neighbor region ($k = 8$) and repeatedly perform a max-reduce over a local sensor reading. An additive-increase/additive-decrease controller is used, which takes an exponentially-weighted moving average (EWMA) of the yield of each reduction operation. If the yield is 10% greater than the target, the maximum retransmission count is reduced by one; if it is 10% less than the target, the count is increased by one.

This simple algorithm is very effective at meeting a given yield target, as shown in Figure 8. Note that for targets below 0.5, the controller overshoots the target somewhat. This is mainly because most nodes are well-connected, and even a low retransmission count will result in a good fraction of messages getting through. Another interesting metric is the average number of messages exchanged for each reduction operation (not shown in the figure). Reductions are implemented by the root sending a request message to each neighbor in the region, which replies with the value of the requested shared variable. Therefore, with no message loss, two messages are exchanged per node. As the yield target scales, this number ranges from 3 (for a yield target of 0.2) to about 10.5 (for a target of 0.9), which gives an indication of the additional overhead for achieving a target reliability.

6.2 Approximate planar mesh construction

Constructing an approximate planar mesh is a tradeoff between the number of messages sent and the quality of the resulting mesh, which we measure in terms of the fraction of crossing edges. Given the unreliable nature of the communication channel, our pruned Yao graph algorithm cannot guarantee that the mesh will be planar. For many applications, a perfect mesh is not necessary, since planarity is impossible to guarantee if there is measurement error in node localization. As we will see in the next subsection, the quality of the planar mesh has a direct influence on the accuracy of contour detection.

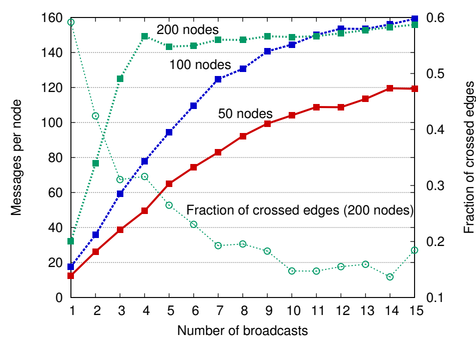


Figure 10: **Quality and overhead of the pruned Yao graph as a function of number of node advertisements.** *The fraction of crossed edges is substantially similar for 50 and 100 nodes.*

In our implementation, the fraction of crossed edges in the mesh depends primarily on the number of broadcasts made by each node to advertise its location. These advertisements are used to form the k-nearest-neighbor region, the first step in the planar mesh construction. Intuitively, hearing from a greater number of radio neighbors allows the k-nearest-neighbor region to select the closest neighbors from this set. Missing an advertisement may cause nodes to select more distant neighbors, leading to a larger proportion of crossed edges.

Figure 10 shows the cost in terms of the number of messages sent per node, as well as the fraction of crossed edges, as the number of node advertisements is increased. Note that the message overhead does not grow linearly with the broadcast count; this is because we are counting all messages involved in mesh construction, including exchanging location information, outedge advertisements, and edge invalidation messages. There is a clear relationship between increased communication, and hence increased bandwidth and energy usage, and the quality of the mesh. As in the case of reduction, applications can tune the number of broadcasts to meet a given target resource budget or mesh quality.

The effect of increasing network density is also shown in the figure. In all cases, nodes are distributed over a fixed geographic area. As density increases, so does the message overhead, since nodes have more neighbors to consider during graph construction.

6.3 Contour finding accuracy

Next, we evaluate the contour finding application, described in Section 5, in terms of the quality of the underlying approximate planar mesh. We characterize the accuracy of contour finding as the mean error between each computed contour point and the actual contour location. Crossed edges in the mesh are likely to introduce error in the contour calculation, as they generally connect two nodes that may not be closest to the actual contour.

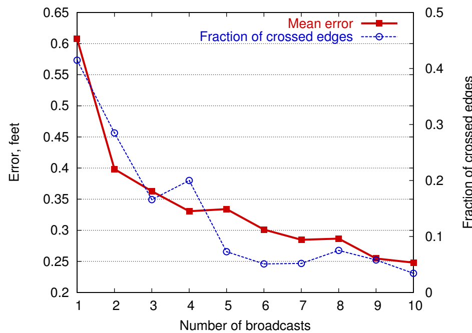


Figure 11: **Accuracy of contour detection tracking as a function of the number of node advertisements.**

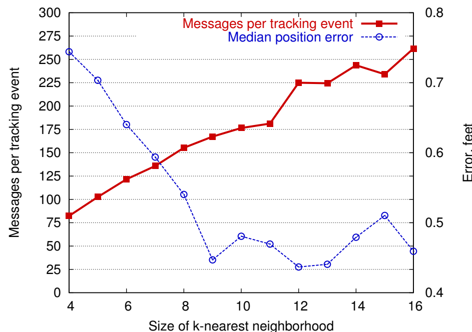


Figure 12: **Accuracy and overhead of object tracking as a function of neighborhood size.** *Results are the average of three runs for each neighborhood size.*

We simulated a linear contour passing through the center of the sensor field that rotates by an angle of 0.1 radians every 5 sec. Nodes take local sensor readings once a second and those that are above the sensor threshold compute new contour points. This scenario stresses the contour finding application and causes all areas of the network to eventually calculate contour points as the frontier rotates. In each case we ran the application for 100 simulated seconds.

Figure 11 shows the error in contour detection as the number of broadcasts used to construct the planar mesh is increased. There is a clear relationship between the overhead of mesh formation and the accuracy of contour detection. Also shown is the fraction of crossed edges, following a pattern similar to that in Figure 10.

6.4 Object tracking accuracy

Next, we evaluate the accuracy of the object tracking application described in Section 5. The application tracks a simulated object moving in a circular path of radius 6 feet at a rate of 0.6 feet every 2 sec. As with contour finding, moving the object in a circular path causes nodes in different regions of the network to detect and track the object. Nodes take sensor readings once a second, the value of which scales linearly with the node's distance to the object, with a maximum detection range of 5 feet. In each case we run the application for 100 simulated seconds and average over three runs.

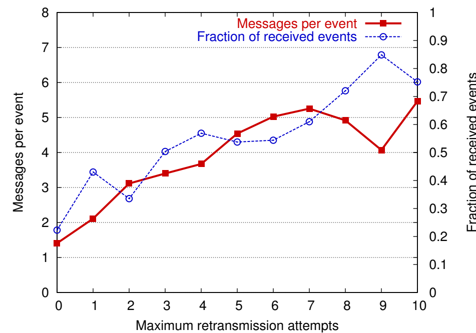


Figure 13: **Reliability and overhead of event detection as a function of retransmission attempts.**

The resource tuning parameter in question is the size of the k-nearest neighbor region. A smaller number of neighbors reduces communication requirements, but yields a less accurate estimate of the object location. Note that increasing the neighborhood size beyond a certain point will not increase tracking accuracy, as more distant nodes are less likely to respond to the reduction request due to packet loss.

Figure 12 shows the accuracy of the tracking application as we vary the number of neighbors in each region. For each time step, we calculate the distance between the simulated object and the value reported by the sensor network. As the figure shows, as the size of the neighborhood increases, so does the accuracy, as well as the number of messages sent (network-wide) per tracking event. Above about $k = 8$, the increase in the neighborhood size does not improve performance appreciably, in part because the additional nodes are further from the target object and may not detect its presence. Also, the more distant nodes may have poor radio connectivity to the root, as described above.

6.5 Event detection reliability

Finally, we evaluate the use of directed diffusion, implemented using the spanning tree region, to detect the presence of a moving object through the sensor field. The base station, located in the upper-left corner of the sensor field, forms a spanning tree region and floods interests to the network through the directed diffusion interface. Rather than computing the location of the object, nodes with a local sensor reading over a fixed threshold (corresponding to an object distance of 2.5 feet) route an event report back to the base. We compute the total number of messages as well as the fraction of event reports received at the base station as the number of retransmission attempts is scaled.

Figure 13 presents the results in terms of the number of messages transmitted per detected event, as well as the fraction of events received by the base station. As with the other examples, scaling the maximum retransmission

count increases the number of messages, and has a corresponding effect on the reliability if event delivery to the base. In all of these cases, the core benefit of the abstract regions interface is the ability to tune the underlying communication substrate to achieve resource management and accuracy goals.

7 Conclusions and Future Work

As sensor networks become more common, better tools are needed to aid the development of applications for this challenging domain. We believe that programmers should be shielded from the details of low-level radio communication, addressing, and sharing of data within the sensor network. At the same time, the communication abstraction should yield control over resource usage and make it possible for applications to balance the tradeoff between energy/bandwidth consumption and the accuracy of collective operations.

The abstract region is a fairly general primitive that captures a wide range of communication patterns within sensor networks. The notion of communicating within, and computing across, a neighborhood (for a range of definitions of "neighborhood") is a useful concept for sensor applications. Similar concepts are evident in other communication models for sensor networks, although often exposed at a much higher level of abstraction. For example, directed diffusion [18] and TinyDB [25] embody similar concepts but lump them together with additional semantics. Abstract regions are fairly low-level and are intended to serve as building blocks for these higher-level systems.

We have described several abstract region implementations, including geographic and radio neighborhoods, an approximate planar mesh, and spanning trees. The implementation of abstract regions in the TinyOS environment relies on fibers, a lightweight concurrency primitive that greatly simplifies application design. Finally, we have evaluated the use of abstract regions for four typical sensor network applications: object tracking, contour finding, distributed event detection, and geographic routing. Our results show that abstract regions are able to provide flexible control over resource consumption to meet a given latency, accuracy, or energy budget.

In the future, we intend to explore how far the abstract region concept addresses the needs of sensor network applications. We are currently completing a suite of abstract region implementations and are developing several applications based on them. We also intend to provide a set of tools that allow application designers to understand the resource consumption and quality tradeoffs provided by abstract regions. These tools will provide developers with a view of energy consumption, communication overheads, and accuracy for a given application. Our goal is to allow designers to express tolerances (say, in terms of a resource budget or quality threshold) that map onto the tuning knobs offered by abstract regions. This process cannot be performed entirely off-line, as resource requirements depend on activity within the network (such as the number of nodes detecting an event). Runtime feedback between the application and the underlying abstract region primitives will continue to be necessary.

Our eventual goal is to use abstract regions as a building block for a high-level programming language for sensor networks. The essential idea is to capture communication patterns, locality, and resource tradeoffs in a high-level language that compiles down to the detailed behavior of individual nodes. Shielding programmers from the details of message routing, in-network aggregation, and achieving a given fidelity under a fixed resource budget should greatly simplify application development for this new domain.

Acknowledgements

The authors wish to thank Ion Stoica for his excellent advice in shepherding this paper, as well as the comments by the anonymous reviewers. Nelson Lee and Phil Levis provided much of the simulation framework used in the analyses. We also wish to thank Kamin Whitehouse and Cory Sharp for their comments on the neighborhood abstraction, as well as the tracking application upon which our algorithm is based.

References

[1] S. Adlakha, S. Ganeriwal, C. Schurgers, and M. B. Srivastava. Density, accuracy, latency and lifetime tradeoffs in wireless sensor networks - a multidimensional design perspective. In review, 2003.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management, or, event-driven programming is not the opposite of threaded programming. In *Proc. the USENIX 2002 Annual Conference*, June 2002.

[3] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *Proc. the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[4] A. Boulis, S. Ganeriwal, and M. B. Srivastava. Aggregation in sensor networks: An energy - accuracy tradeoff. In *Proc. IEEE workshop on Sensor Network Protocols and Applications*, 2003.

[5] R. Brooks, P. Ramanathan, and A. Sayeed. Distributed target classification and tracking in sensor networks. *Proceedings of the IEEE*, November 2003.

[6] Center for Embedded Network Sensing. Contaminant transport monitoring. `http://cens.ucla.edu/Research/Applications/ctm.htm`.

[7] Center for Information Technology Research in the Interest of Society. Smart buildings admit their faults. `http://www.citris.berkeley.edu/applications/disaster_response/smartbuil%dings.html`, 2002.

[8] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proc. the Workshop on Data Communications in Latin America and the Caribbean*, Apr. 2001.

[9] R. X. Cringely. Chase Cringely: Finding Meaning in a Lost Life. `http://www.pbs.org/cringely/pulpit/pulpit20020425.html`.

[10] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An evaluation of multi-resolution search and storage in resource-constrained sensor networks. In *Proc. the First*

ACM Conference on Embedded Networked Sensor Systems (SenSys 2003), November 2003.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. Programming Language Design and Implementation (PLDI)*, June 2003.

[12] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. DIFS: A distributed index for features in sensor networks. In *Proc. the First IEEE International Workshop on Sensor Network Protocols and Applications*, May 2003.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, 1994.

[14] J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proc. the 18th SOSP*, Banff, Canada, October 2001.

[15] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. the 5th ACM/IEEE Mobicom Conference*, August 1999.

[16] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *Proc. the 2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, March 2003.

[17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Boston, MA, USA, Nov. 2000.

[18] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. International Conference on Mobile Computing and Networking*, Aug. 2000.

[19] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, Boston, MA, August 2000.

[20] V. A. Kottapalli, A. S. Kiremidjian, J. P. Lynch, E. Carryer, T. W. Kenny, K. H. Law, and Y. Lei. Two-tiered wireless sensor network architecture for structural health monitoring. In *Proc. the SPIE 10th Annual International Symposium on Smart Structures and Materials*, San Diego, CA, March 2000.

[21] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[22] D. Li, K. Wong, Y. H. Hu, and A. Sayeed. Detection, classification and tracking of targets in distributed sensor networks. *IEEE Signal Processing Magazine*, 19(2), March 2002.

[23] X.-Y. Li, P.-J. Wan, Y. Wang, and O. Frieder. Sparse power efficient topology for wireless networks. In *Proc. 35th Annual Hawaii International Conference on System Sciences*, January 2002.

[24] J. Liu, P. Cheung, L. Guibas, and F. Zhao. A dual-space approach to tracking and sensor management in wireless sensor networks. In *Proc. the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, Atlanta, Georgia, September 2002.

[25] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. the 5th OSDI*, December 2002.

[26] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. the ACM SIGMOD 2003 Conference*, June 2003.

[27] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, Sept. 2002.

[28] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An architecture for enabling sensor-enriched Internet service. Technical Report IRP-TR-03-04, Intel Research Pittsburgh, June 2003.

[29] K. S. Pister. Tracking vehicles with a uav-delivered sensor network. `http://robotics.eecs.berkeley.edu/~pister/29Palms0103/`, March 2001.

[30] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage in sensornets. In *Proc. the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, Atlanta, Georgia, September 2002.

[31] J. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1-3):21–74, May 2002.

[32] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *Proc. the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[33] M. Welsh. Exposing resource tradeoffs in region-based communication abstractions for sensor networks. In *Proc. the 2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[34] M. Welsh, D. Myung, M. Gaynor, and S. Moulton. Resuscitation monitoring with a wireless sensor network. In *Supplement to Circulation: Journal of the American Heart Association*, October 28, 2003.

[35] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. the International Conference on Mobile Systems, Applications, and Services (MOBISYS '04)*, June 2004.

[36] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[37] Y. Xu and W.-C. Lee. On localized prediction for power efficient object tracking in sensor networks. In *Proc. 1st International Workshop on Mobile Distributed Computing*, May 2003.

[38] Y. Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3), September 2002.