EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks *

Abdelzaher T., Blum B., Cao Q., Chen Y., Evans D., George J., George S., Gu L., He T., Krishnamurthy S., Luo L., Son S., Stankovic J., Stoleru R., Wood A.
Department of Computer Science, University of Virginia, Charlottesville, VA 22904

Abstract

Distributed sensor networks are quickly gaining recognition as viable embedded computing platforms. Current techniques for programming sensor networks are cumbersome, inflexible, and low-level. This paper introduces EnviroTrack, an object-based distributed middleware system that raises the level of programming abstraction by providing a convenient and powerful interface to the application developer geared towards tracking the physical environment. EnviroTrack is novel in its seamless integration of objects that live in physical time and space into the computational environment of the application. The performance of an initial implementation of the system is evaluated on an actual sensor network based on MICA motes. Results demonstrate the ability of the middleware to track realistic targets.

Keywords: sensor networks, programming paradigms, tracking, QoS, distributed systems

1 Introduction

The work reported in this paper is prompted by the increasing importance of large-scale wireless sensor networks [15] as a future platform for a growing number of applications such as habitat monitoring [7, 21], intrusion detection [29], defense, and scientific exploration. Advances in hardware miniaturization [10] have made it economically viable to develop embedded systems of massively distributed disposable sensor nodes, characterized by coordination of a very large number of tiny wireless computing elements. A great impediment to rapid deployment of such systems lies in the lack of distributed software and programming support for sensor network applications. A new distributed computing paradigm is needed that exports appropriate abstractions and implements efficient information management protocols in large-scale sensor networks. EnviroTrack is an attempt to develop such a paradigm.

EnviroTrack is a middleware layer that exports a new address space in the sensor network. In this space, physical events in the external environment are the addressable entities. This type of addressing is convenient for applications that need to monitor environmental events. For example, a surveillance application that monitors vehicle movement behind enemy lines may assign unique labels to individual vehicles. Their state can then be addressed by reference to these labels. Moreover, computing or actuation objects can be attached to individual addresses in much the same way computation is assigned to IP hosts in an Internet-like environment. Such attached computation or actuation is then performed in the physical neighborhood of the named entity. Hence, for example, a microphone could be turned on at some network address (e.g., one that names a vehicle in the external environment) to listen-in on the corresponding environmental object. As the named vehicle moves, the middleware will turn on the appropriate nearby node microphones such that a non-interrupted audio stream is delivered to the receiver despite the mobile nature of the source. Communication can also occur between two mobile endpoints. For example, a walking soldier with a PDA may track the position of a suspect vehicle detected elsewhere in the network. In short, we (i) export a novel logical address space in which external environmental objects are the labeled entities, and (ii) allow arbitrary data, computation, or actuation to be attached to such logical network addresses. These data, computation, and actuation are encapsulated in an abstraction we call *tracking objects*.

The EnviroTrack middleware library implements a set of protocols that off-load from an application developer the details of inter-object communication, object mobility, as well as the maintenance of tracking objects and their state. It abstracts away the fact that computation associated with the object may be distributed and performed by all sensor nodes in the vicinity of the tracked physical entity. As the tracked entity moves, the identity and location of the sensor nodes in its neighborhood change, but the tracking object representing it remains the same. The programmer thus interacts with a changing group of sensor nodes through a simple, uniquely addressable, object interface.

EnviroTrack has been implemented and tested on a popular sensor network platform based on MICA motes [16]. Our initial implementation of this infrastructure uses compiled NesC [13] programs on TinyOS [15], an operating system for

*The work reported in this paper was supported in part by the National Science Foundation grants EHS-0208769, CCR-0205327, and CCR-00-92945, DARPA grant F33615-01-C-1905, and MURI grant N00014-01-1-0576.

sensor networks. Recent advances in programming support for sensor networks, such as the development of a virtual machine [19], will significantly simplify the code development and dissemination effort in the future. We present evaluation results, which illustrate how typical sensor-network applications that use EnviroTrack will perform on the current hardware platform.

The rest of this paper is organized as follows. Section 2 defines the tracking problem in more detail, describes our programming system architecture, and elaborates on the main abstractions provided by EnviroTrack. Section 3 illustrates how a sample tracking application can be written in EnviroTrack. Section 4 provides implementation details. Section 5 presents a performance evaluation. An overview of related work is presented in Section 6. The paper concludes with Section 7.

2 Programming Model

The programmer's view of an application written in EnviroTrack is depicted in Figure 1. Sensors which detect certain user-defined entities in the physical environment form groups, one around each entity. A network abstraction layer associates a *context label* with each such group to represent the corresponding tracked entity in the computing system. Context labels can be thought of as logical addresses of virtual hosts (contexts) which follow the external tracked entity around in the physical environment. In the following, we use contexts and context labels interchangeably. Objects can be attached to context labels to perform context-specific computation. These attached objects are called *tracking objects*. They are executed on the sensor group of the context label. Since the actual location of the tracking object is the nodes in the physical vicinity of the target, the object can perform local sensing and actuation to interact directly with the target's locale. For completeness, EnviroTrack also supports conventional static objects that are not attached to context labels.

Context labels have types depending on the entity tracked. For example, a context label of type CAR is created wherever a car is observed. To declare a context label of some type e (named after the tracked event type), the programmer must supply three pieces of information. First, the programmer supplies a function $sense_e()$ that describes the sensory signature identifying the tracked environmental target. For example, if the context type is to identify moving vehicles, $sense_e()$ might be a function of magnetometer and motion sensor readings. The middleware watches for the specified sensory pattern in the environment and creates a sensor group around the detected target when the pattern occurs. This function is also used to maintain the membership of the sensor group around the tracked target when the target moves. Group membership, in this case, is restricted to those nodes that sense the given target (i.e., for which $sense_e()$ is true).

Second, the programmer declares what constitutes the environmental state to be encapsulated in the context label. This

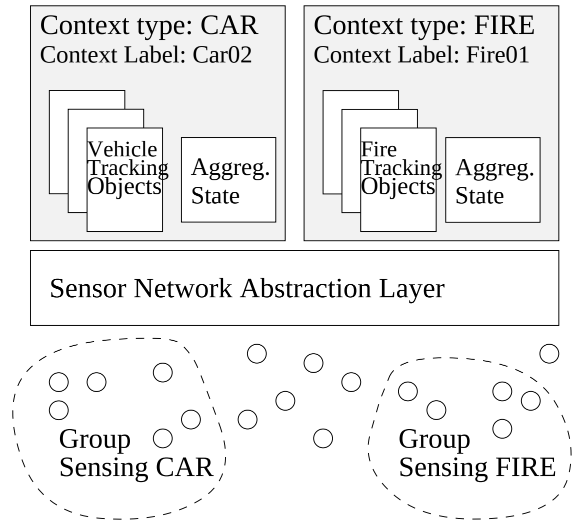

Figure 1. Programming Model

state is shared by all tracking objects attached to this label. State is declared by defining an aggregation function $state_e()$ that acts on the readings of all sensors for which $sense_e()$ is true or was true within a recent past defined by a freshness constraint. The aggregation is carried out locally by a sensor node that acts as the group leader of all sensors sensing the named target. The aggregation function can also include a critical mass constraint that specifies the minimum number of sensors that must be involved in the aggregation for the result to be statistically meaningful. EnviroTrack provides a library of the most common distributed aggregation functions to choose from, such as addition, averaging, and median computation. These functions can also be location-aware, for example, to compute the center of gravity of the measurements. The underlying infrastructure includes a data collection protocol executed by the leader to collect, timestamp, and log sensor data (i.e., the arguments for the $state_e()$ function) from sensor group members satisfying $sense_e()$. The $state_e()$ function is then applied on the collected data in a way that satisfies the freshness and critical mass conditions. Finally, the programmer specifies which objects are to be attached to the context label. Attached object code can reference the aggregate state maintained by the leader in this context.

In the following, we describe in more detail the network abstraction layer, tracking objects, and aggregate state.

2.1 Network Abstraction Layer

Context labels abstract sensor groups for the programmer. The programmer is aware that a distributed computation, associated with the context label, is executed on multiple sensors in the vicinity of a tracked entity. The programmer, however, is not involved in managing the membership, leader election, and leader handoff in the sensor group.

A sensor node joins the sensor group of a particular context when its local sensor readings satisfy the boolean condition $sense_e()$. It leaves the group when this condition is no longer satisfied.[1] A sensor node can be part of multiple groups at one time. Programs running for different groups are effectively independent. The sensor group associated with a context label maintains two invariants. First, all members of a group at time *t* satisfy the condition $sense_e()$. Second, the group is not partitioned. All members of a sensor group can communicate with each other possibly using multiple hops through other members of the same group. This physical continuity constraint is introduced to ensure that groups formed around different entities of the same type remain distinct and do not merge as long as the tracked entities are physically separated.

2.2 Tracking Objects

The tracking objects attached to a context label consist of methods that are invoked either by the passage of time (time-triggered), or by the arrival of messages that carry method invocation requests. Object code is executed on a single node. In the current implementation, this node is the sensor group leader of the enclosing context. Object code may make references to the aggregate state maintained by the enclosing context, returned by the $state_e()$ function. This state is collected by a distributed data collection protocol which constitutes the distributed part of the objects' computation. Note that the code is independent of the number and identity of participants of the distributed data collection protocol. It can assume, however, that the aggregation results always satisfy the semantics of aggregate state (i.e., they are in accordance with the specified freshness and critical mass requirements).

2.3 Aggregate State

The function $state_e()$ is configured by declaring aggregate state variables for context e. The definition of a state variable in the enclosing context specifies three important pieces of information:

- Aggregation function. Aggregation functions produce scalar values from sets of sensor readings. Several aggregation functions are provided in a library that can be extended by the programmer.

- Freshness L_e. The freshness threshold tells the system how long sensor readings can be used before they are considered stale. Only readings taken within the prescribed freshness time are used to compute the value of an aggregate state variable.

- Critical mass N_e. The critical mass is an integer that denotes the minimum number of sensor nodes that should be involved in the aggregation for the returned value to be valid. Only readings produced within the freshness threshold can contribute to the critical mass threshold.

Since freshness is decided at configuration time, nodes that join the group associated with a particular context label periodically send to the leader their measurements at a period $P_e = L_e - d$, where d is an estimate of maximum message delay and processing time within the group. This ensures that the results of aggregation are always based on sensor readings that are not older than L_e. The leader maintains approximate aggregate state by performing the aggregation function periodically on all the messages received within a sliding window of P_e time units. The state is tagged valid (using a *valid* flag) if more than N_e messages were received within the window. The application code running on the leader, can perform asynchronous *read* operations on aggregate state variables, which return their current value and validity status.

Figure 2 shows the overall internal structure of the middleware, illustrating both member and leader code. As seen in figure, the main function of members is to report their readings periodically to the group leader. The leader computes the aggregate state and runs the application, which may communicate with remote contexts using a message transport protocol. A distributed group management protocol keeps track of group membership and leader election. Observe that each sensor node has both member and leader code. The role taken by the node is chosen by the group management protocol.

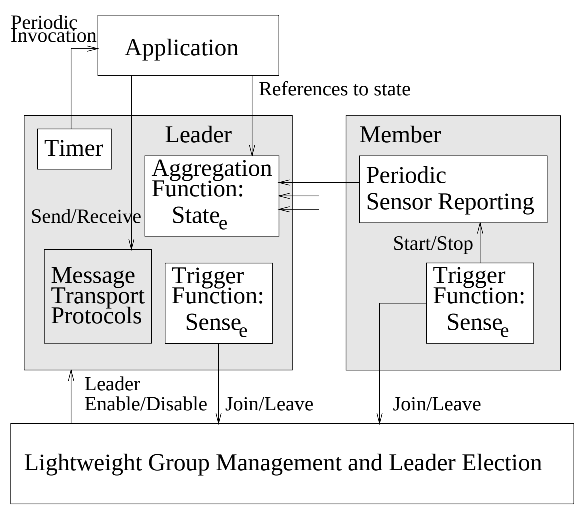

Figure 2. Middleware Architecture

3 Language Features and Application Example

To facilitate the use of our middleware, we developed simple language support for declaring context labels and aggregate

[1] Alternatively, a separate deactivation condition may be written.

```
(1)  begin context tracker
(2)    activation: magnetic_sensor_reading()
(3)    location : avg (position) confidence=2, freshness=1s
(4)
(5)    begin object reporter
(6)      invocation: TIMER(5s)
(7)      report_function() {
(8)        MySend (pursuer, self.label, location);
(9)      }
(10)   end
(11) end context
```

Figure 3. Sample EnviroTrack Code

state variables. A preprocessor uses the stated declaration to emit appropriate code that initializes the middleware and configures the $state_e()$ and $sense_e()$ functions. The preprocessor then configures the trigger conditions for membership in particular contexts, and replaces references to the aggregate state variables by middleware function calls that evaluate and return them at runtime.

An EnviroTrack program consists of a list of context declarations such as the one shown in Figure 3. Each context declaration includes an *activation* statement specifying the $sense_e()$ condition for creating new instances of the declared context type. The activation statement is followed by aggregate state declaration for the created context. This declaration consists of a list of variables, each with its own freshness and critical mass constraints. The declared aggregate state variables are computed for the context at run-time as described in Section 2.3. This computation is performed independently of application code. Finally a list of objects is attached. Each object may have NesC functions with optional *invocation* conditions. Invocation conditions may be written in terms of aggregate state variables defined in the enclosing context. They state when the particular method is to be invoked. All static objects are declared separately within the *default* context type.

We illustrate our programming syntax by an application example. A typical sensor network application is one in which a dense network of motes is deployed to track the location of moving vehicles. For simplicity of illustration, we assume that the presence of the vehicle is determined using a magnetic sensor. In our application, sensors that detect magnetic distortion caused by the vehicle form a group abstracted by a context label. Note that several context labels may be instantiated, depending on the number of vehicles sensed. In each context label, the attached object periodically reports the vehicle's location to a preselected mote interfaced to a mobile pursuer. The pursuer (a laptop) monitors all vehicles at all times and records their tracks. The program in Figure 3 shows how the vehicle-tracking context is defined. Pursuer code is not shown.

The example in the figure defines a context of type *tracker*. Line 2 specifies that the activation condition, $sense_e()$, for this context type is encoded by the boolean function *magnetic_sensor_reading()*. This function is written in NesC. It returns a true value when a vehicle is detected. Line 3 defines one aggregate variable, namely, the average position $location$. It specifies that the value of $location$ returned upon a reference must represent the average of at least 2 sensor node readings measured no earlier than 1 second ago. Hence, $N_e = 2$ and $L_e = 1$.

Lines 5-10 describe the attached computation. Line 6 specifies when the computation is invoked. It dictates that the report function be invoked periodically with a period of 5 seconds. This is followed by the code of the function. This code simply makes a call to *MySend()* which in turn calls the routing layer to send the message to the pursuer. Two parameters are passed in the message, a handle of the originating context label obtained using $self.label$ and the aggregate state variable $location$ indicating the average position of all sensors currently detecting the reported vehicle (i.e., the estimated position of the vehicle).

The above code will generate multiple instances of the tracker if multiple vehicles are present. Further, even though the vehicles move and the sensor nodes comprising their corresponding contexts will change, the context labels will not. This significantly simplifies the programmer's interaction with the varying sensor group tracking each vehicle.

4 Implementation

In this section, we describe implementation issues in EnviroTrack. Our implementation is built on TinyOS [15], an operating system kernel developed exclusively for sensor nodes. TinyOS provides support for communication, multitasking, and code modularity. Geared towards communication-intensive applications, it exports the abstraction of components, which can be integrated into structures similar to a protocol graph. Each component consists of command handlers, event handlers and simple tasks. Communication protocols can be constructed easily in a modular manner by developing the appropriate handlers independently of others. The implementation of the EnviroTrack programming system consists of the following main modules:

- The EnviroTrack preprocessor: This preprocessor translates EnviroTrack declarations such as the one shown in Section 3 into NesC code which calls run-time libraries implementing group management, data aggregation and communication.
- The group management protocol: This protocol maintains the membership of the sensor group associated with a single context label.
- Routing services: These services implement a communication protocol between different context labels.

These modules are described next.

4.1 The EnviroTrack Preprocessor

The input to the EnviroTrack preprocessor is the context description file, such as the one shown in Section 3. The preprocessor patches a set of NesC program templates using the information gathered from the context description file to produce the target NesC modules such as those implementing the $sense_e()$ and $state_e()$ functions. The programs are then compiled using the provided TinyOS development tools.

The outer loop of our TinyOS program template code is implemented as a timer handler. This handler is invoked on the sensor group leader periodically and executes one iteration per invocation. The handler maintains an array of contexts. Each entry represents one context and provides access (via function pointers) to that context's activation condition, $sense_e()$, and object code, as well as its status. The generic handler in the template simply goes through this array checking if any context satisfies the activation condition. The compiler emits an `initContextStructures()` function that sets up this array based on the context description file. At run-time, sensor devices remain in this time-triggered mode until an appropriate condition is sensed. Activation conditions of different contexts are expressed in terms of boolean NesC functions which access local sensory measurements. These functions are sensor dependent. They can be written by the developer or chosen from a common library.

When an activation condition, $sense_e()$ is satisfied for a context of type e, group management services are activated on the motes sensing that condition. The execution of these services creates a context label (of type e) and maintains its approximate aggregate state, $state_e()$, on the current group leader. Subsequent invocations of the timer handler check for method invocation conditions defined in terms of this aggregate state, and post TinyOS tasks to execute methods whose invocation conditions are satisfied.

In the current implementation, objects are permanently attached to contexts. Each of the methods attached to a context is emitted with their names mangled (by adding the context name). The contents of each function are also parsed to replace references to aggregate variables with function calls that return the aggregate variable's value in accordance with its specified tracking QoS. Every possible aggregation for every sensor value is available as a function call. The naming of these functions is done based on a known scheme so as to allow the compiler to generate the correct call. Each aggregate variable is associated with attributes of freshness and critical mass. The functions (that return aggregate values) themselves are patched with the right value of freshness and confidence to produce the specified QoS.

4.2 Group Management Services

Group management services, shown at the bottom of Figure 2 maintain coherence of context labels. That is, they ensure that a group of sensors identifying the *same entity* in the environment produce a *single* context label. This label must persist and remain unique even as the membership of this sensor group changes. Ideally, to maintain context label coherence, at any point in time, nodes sensing the same external entity maintain a single "majority" leader.

Contexts are created when a node first senses condition $sense_e()$. The node immediately starts a leader election process in which it randomly chooses a small timeout value. A node which times out first sends a message informing its neighbors that it is leader. Upon receipt of this message, other nodes sensing the same $sense_e()$ condition become members. We require that a node's communication radius be larger than twice its sensing radius such that all nodes sensing the same target are within each other's communication range.

An elected group leader sends periodic heartbeats, which are received by all group members. Leader heartbeats have three purposes. First, they inform current members that the leader is alive. Should the leader die, a new leader election is started after a timeout. Second, they carry application state that must persist across leader handoffs. This state is recorded by all member nodes. This mechanism allows new leaders to continue computations of failed leaders from the last state received. An application can explicitly create persistent state using a $setState()$ primitive and read it using $getState()$. Finally, heartbeats are overheard past the group's perimeter thus informing neighboring nodes of the existence of context label e. Nodes that cannot sense the target themselves but know of its existence from nearby leader heartbeats are called *group followers*. If these nodes subsequently sense the condition $sense_e()$, they join the present group instead of forming a new context label. The mechanism ensures that multiple spurious context labels do not emerge around the same target. When the leader gets out of sensory range from the target, it sends a leader handoff message which initiates a new leader election. The resulting behavior is that a group with a unique leader is created around each target. Membership changes and leader (and state) handoffs occur automatically as the target moves.

A detailed simulation study of the above protocol appeared in [4] in which particular attention was paid to various failure and message loss scenarios that result in election of spurious leaders. It was shown that while spurious leaders do emerge, very simple techniques can substantially reduce their effect on system behavior. For example, in the presence of message loss, a leader handoff may produce two nodes both of whom claim to be leaders of the same context label. However, since these nodes are within each other's communication range, the one with the higher node identifier can eventually force the other to relinquish leadership. The same applies if a node elects itself as leader of a new context label for a target that is already being tracked by another. The effect of such spurious context labels is reduced by letting nodes that hear two nearby leaders ignore the one with the smaller *weight*. Each new context label is initially created with a leader weight of zero. Leaders of existing context labels accrue a weight equal to the number of messages received by the leader from members to date. This weight is passed during leadership handoffs. Hence, leaders of spurious context labels will generally be ignored. Consequently, the abstraction of a single context label per target is adequately maintained.

The mechanism described above opens several important questions for future research. One is what do when multiple targets cross paths. In the present scheme a violation of context label coherence may occur. For example, the "younger" context label may disintegrate (be absorbed in the group of the "older") and later emerge as a different label when the targets separate. Such anomalies should be dealt with at the application layer. It may be impossible to solve them in middleware without complex signal processing as it may be impossible, say, for a magnetic sensor to identify which of two nearby targets is responsible for its magnetic reading. From the application's perspective, the sensor network has a notion of granularity which defines the resolution of target detection and is related to the communication radius of nodes. If multiple targets fall within the same granule, they become indistinguishable. When they separate, they again become distinct targets.

4.3 Routing Services

To route among different context labels, we use an algorithm similar to landmark routing [22]. Nodes are assumed to know their location such that geographic routing can be used. Leaders of established context labels who wish to communicate broadcast their existence and report their location to a landmark. Other nodes route packets to the landmark, which in turn forwards them to the leader of the context label. Upon leader handoffs (the location of) the new leader is reported to the landmark. In addition, a forwarding pointer is inserted at the previous leader to forward packets that are in transit. On top of the routing layer a simple demultiplexor is implemented that dimultiplexes incoming messages at the destination and forwards them to one of several application modules. This allows implementing remote method invocation. The destination address of the remove method contains the name of the context label and the method identifier. The latter is used by the demultiplexor to identify the module implementing the needed method.

5 Performance

In this section, we evaluate the performance of an actual implementation of the presented tracking middleware service. The implementation is on MICA motes running TinyOS. While some simulation studies have been performed on the group management protocol [4] as mentioned in Section 4.2, this is the first detailed report on the performance of an actual implemented prototype of the complete service. In the context of performance evaluation, it is interesting to node that the programming interface imposed on top of our middleware does not interfere with its run-time performance. In fact, this interface was written by the authors after the tracking middleware was developed. It simply automates the process of configuring the middleware for tracking. Once the preprocessor has parsed the user's context declarations and emitted the configured code, the middleware looks the same as if it was hand-coded. No performance penalty is associated with the improved level of abstraction.

With the above observation in mind, we now present the experimental performance of tracking. We first establish a case for the viability of our middleware for tracking in practice. We then proceed with stress-testing EnviroTrack to explore the limitations of the current implemented prototype.

5.1 A Case for Tracking

Our case-study target is the T-72 tank (made in Russia), moving in an off-road sensor field. This particular tank weighs 44 tons and has a maximum off-road speed of around 45 km/hr [12]. Sensors in the field are equipped with magnetometers. Honeywell advertises magnetic traffic monitoring sensors which can detect moving vehicles from a range of up to 30 meters [20]. These sensors operate by detecting slight disturbances to the Earth magnetic field caused by ferrous objects. The magnitude of this disturbance depends on the amount of the ferrous material in the tracked object. Since the T-72 tank weights about 40 times the average vehicle in ferrous matter, its presence could be detected at a much larger distance than 30 meters. Magnetic effects are attenuated with the cube of the distance. Hence, we set the magnetic detection radius for the tank to approximately $30 * 40^{1/3}$ which amounts to about 100 meters. It is easy to show geometrically that if the tank can be detected 100 meters away, it is guaranteed that it is always within range from at least one sensor as long as sensors are put on a grid about 140 meters apart. We thus assume a rectangular grid of sensors with a per-hop distance of 140 meters. Note that covering a border area of say 70 km x 5 km at this spacing would require roughly 18,000 sensor devices, which is about the right size for the envisioned sensor networks. Moving at its maximum speed, a T-72 tank will cover one hop every 11.2 seconds.

We developed a testbed which provides a scaled down, 1000:1, model of this scenario. To experiment with variable sensor range more readily, we replaced magnetic sensors with light sensors installed on MICA motes. The magnetic field of the target was emulated by moving a round object of a corresponding radius above the sensor field to block a strong light source from the appropriate sensors. The field was arranged into a rectangular grid. In our first experiment, the tracked object was moved at a speed of 10 seconds/hop and 15 seconds/hop, which corresponds to an emulated speed of 50 km/hr and 33 km/hr, respectively. A single context type was defined, whose declaration is similar to Figure 3. At run-time a context label was generated. Group management maintained a leader for the context label. The leader sent to a base station the average position reported by nodes sensing the target at the current time. After each run, logs on individual motes were inspected to produce message loss and total throughput statistics. Message loss was computed by counting the number of messages sent but never received on any other mote.

Figure 4 shows the real and tracked object trajectory (reported to the base station) in a representative run. The motes were put at integer (x, y) coordinates. The horizontal line at $y = 0.5$ is the real target trajectory. The tracking error occurs because our sensors have no notion of proximity to the target. Moreover, direction anomalies occur due to message loss which causes sensor position aggregation to use a subset of reporting sensors only. An application receiving this trajectory can presumably improve the results by applying filtering to the reported raw data. Results could be further improved if sensor nodes could perform ranging to estimate target proximity.

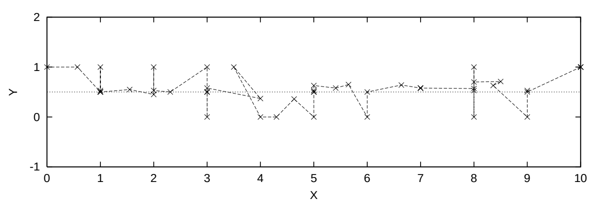

Figure 4. Tracked Tank Trajectory

Figure 5 shows the percentage of successful context label handovers for two target speeds and two settings of group management parameters. A successful handover means that the context label successfully follows tank location by virtue of leadership handoff from one member node to another along the target's path. An unsuccessful handover means a different context label is spawned at the new tank's location, not realizing that it refers to the same tank as the current context label. This case violates context label coherence.

In the first group management parameter setting, leader heartbeats are not propagated past the sensing radius. As expected, in this case it is more likely that multiple context labels are generated for the same target since nodes which sense the target for the first time might not be aware of the existing context label. Figure 5 shows that a fraction of handovers will fail in this case unless target speed is slow. In the second setting, the sensing and communication ranges are such that leader heartbeats are propagated beyond the sensing radius. In this case, all handovers are successful at both emulated tank speeds. This is in agreement with expectations since the group management algorithm in Section 4.2 requires that the communication range be larger than the sensing range. The experiment demonstrates the importance of setting these ranges correctly not to violate the group management assumptions.

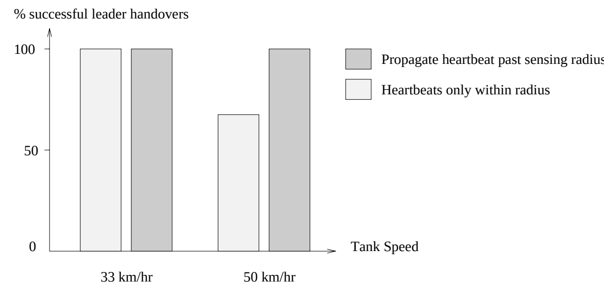

Figure 5. Successful Handovers

Finally, Table 1 shows sample communication data collected during our experiments for the second (correct) case above. Each point is averaged over three independent runs. In particular, we show the measured percentage of lost leader heartbeats (HB loss), lost sensor messages incurred during data aggregation (Msg loss), and the average useful link utilization (Link Util). To compute the latter, we divided the total number of bits sent per second by the total link capacity (50kbs for MICA motes). Hence, this is a worst case estimate, since it assumes a broadcast model in which no two messages could be sent concurrently.

The table demonstrates four important points. First, our system operates correctly in the presence of message loss, which is necessary in sensor network applications. Second, message loss is not caused by link utilization, but rather by the unreliability of the wireless medium (no reliability is implemented in the MAC layer of the MICA motes). Note that the effect of collisions increases with target speed. Third, our communication requirements constitute only a tiny fraction of available link capacity. Hence, we have not yet stressed the limits of the system's capabilities. Fourth, link utilization increases only slightly with tank speed. Hence, the bandwidth requirements of the algorithm have potential to scale well with tracking difficulty.

Speed	% HB loss	% Msg loss	% Link Util
33 km/hr	7.08	3.05	2.54
50 km/hr	22.69	17.05	2.88

Table 1. Communication Performance Data

The aforementioned proof-of-concept results show that the severe limitations on the memory, CPU, and network bandwidth of the MICA motes do not prevent them from performing communication protocol stack processing, group management, leader handoff, and aggregate state computation associated with maintaining our context label abstraction. Moreover, with appropriate sensor selection and parameter settings, realistic targets can be successfully tracked. Next, we stress-test the architecture to determine the maximum trackable target speed as a function of various parameter settings of the middleware.

5.2 Testing the Maximum Trackable Speed

The maximum trackable speed refers to the maximum speed a target can have without causing violations of context label coherence. If a target moves too fast it can be detected by nodes who have not yet heard of it, which results in creation of spurious context labels. The most important parameter which affects the maximum trackable target speed in our architecture is the heartbeat period of the group leader. In the experiments conducted, the timeout associated with failed leader detection (due to absence of heartbeats) is set to 2.1 the heartbeat period. In other words, we wait for two consecutive missing heartbeats before initializing leader re-election.

The maximum trackable speed is computed for the worst-case scenario, which is the case when the current leader fails causing leadership takeover to take place. In this case, a slow heartbeat period will allow the target to escape tracking during the leadership takeover. Consequently, several disconnected groups will be formed (as the target is rediscovered independently at different points along its track). The maximum trackable speed (the highest target speed at which the single group abstraction is maintained) observed in the experiment is shown in Figure 6 as a function of heartbeat period for two events: a narrow siganture event (outer bars), and a wide signature event (inner bars). The figure also shows the trackable speed during normal operation in which each leader willingly relinquishes leadership to another as the target moves out of its sensor range. This case is labeled "relinquish" in the figure and shows a maximum trackable speed that is independent of the heartbeat period.

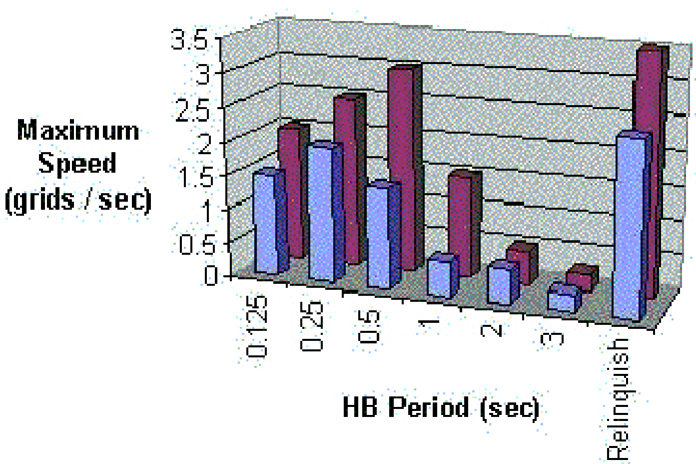

Figure 6. Effect of Timers on Maximum Trackable Speed

Several points can be made from this graph. First, for a large range of parameter settings, the maximum trackable speed is 1-3 hops/s, which is 10-30 times faster than the speed of the tank presented in the previous section. Thus, very fast targets can be tracked, or alternatively, sensors with a much smaller sensing radius can be successfully used to track realistic targets.

Second, we see that events with a larger sensory signature (expressed in figure in terms of multiples of average node separation, or *grids*) can be tracked at higher speeds. This may seem intuitive, as larger targets should be easier to track.

Third, we see that as the heartbeat period is reduced (sending out more frequent heartbeats) faster targets can be tracked. This is intuitive as faster heartbeat makes the group management mechanism more responsive. Realizing that heartbeats are bandwidth-consuming messages and that both CPU and communication bandwidth are limited in our experiments, we stress tested the heartbeat period to determine where overload occurs.

To determine the identity of the bottleneck resource that causes the decline in the maximum trackable speed at small heartbeat periods, we repeated the above experiment in the presence of a substantial amount of cross traffic. The cross traffic was exchanged between motes that do not participate in the EnviroTrack protocol but rather generate "background noise". The shape of Figure 6 in the presence of this cross traffic remained largely unaffected. We therefore conclude that communication bandwidth is not the bottleneck. The bottleneck appears to lie in CPU processing.

In our next experiment, we test the effect of varying the ratio between the communication radius (CR) and the sensing radius (SR) on the trackable target speed. We use explicit leadership handoffs in this experiment (as opposed to handoffs due to leader failures). The results are shown in Figure 7. From this figure, the most important point to note is that for a given CR:SR ratio (which may or may not be a controllable parameter by system designers), larger events are trackable at faster speeds. The direct cause of this is the number of leadership handovers that occur. For a constant speed, when an event is larger, the average time between handovers decreases (as a single leader can sense the target longer) requiring fewer messages to be processed. The lower communication overhead results in a higher trackable speeds. The other point to note is that our tracking architecture breaks down when the CR:SR ratio falls below 1. This occurs because nodes outside of communication range from the leader also sense the event and concurrently form spurious groups thus violating context label coherence. The performance improves as the ratio increases as two nodes that sense the same target are less likely to be outside each other's range.

6 Related Work

A growing challenge facing the distributed systems community is to develop programming paradigms and run-time sup-

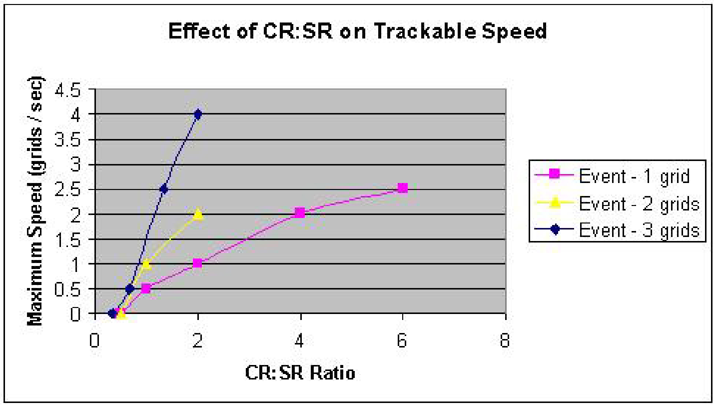

Figure 7. Effect of Sensory Radius on Maximum Trackable Speed

port for the operation of large-scale embedded sensor networks. Classical distributed programming paradigms and middleware such as CORBA [28], group communication [8], remote procedure calls [3], and distributed shared memory [6, 25] share in common the fact that their programming abstractions exist in a logical space that does not represent or interact with objects and activities in the physical world. Their main goal is to abstract distributed communication rather than facilitate distributed sensory interactions with an external physical environment. In contrast, a new paradigm tailored for sensor should be centered around environmentally-driven abstractions aimed at simplifying the coding of interactions with the physical world that arise in distributed deeply embedded systems.

The work reported in this paper is related to several recent projects, such as Cricket [23], Sentient Computing [1] and Cooltown [9], that propose high-level paradigms in which an embedded distributed computing system is able to share perceptions of the physical world. These systems allow the location of entities in the external environment to be tracked. One major difference of these systems from EnviroTrack is that they assume cooperative users who, for example, can wear beaconing devices that interact with location services in the infrastructure for the purposes of localization and tracking [23, 1]. Our interest, in contrast, is in situations where no cooperation is assumed from the tracked entity.

In the absence of cooperation, several research efforts proposed alternative addressing schemes that do not rely on having destinations with specific identities, but rather contact sensor nodes in the vicinity of a phenomenon of interest based on the attributes of data they sense. For example, DataSpace [17] exports abstractions of physical volumes addressable by their locations. Similarly, directed diffusion [18, 14] and the intentional naming system [2] provide addressing and routing based on data interests [18, 14]. Attributed-based naming is also related to the notion of content-addressable networks [24] proposed for an Internet environment, which allows queries to be routed depending on the requested content rather than on the identity of the target machine. We adopt a form of attribute-based naming we call *context labels*. In our architecture, however, context labels are *active* elements. Not only do they provide a mechanism for *addressing* nodes that sense specific environmental conditions, but also they can *host context-specific computation* that tracks the target entity in the environment.

Recent research on system software for sensor networks has seen the introduction of distributed virtual machines designed to provide convenient high-level abstractions to application programmers, while implementing low-level distributed protocols transparently in an efficient manner [27]. This approach is taken in MagnetOS [11], which exports the illusion of a single Java virtual machine on top of a distributed sensor network. The application programmer writes a single Java program. The run-time system is responsible for code partitioning, placement, and automatic migration such that total energy consumption is minimized. Maté [19] is another example of a virtual machine developed for sensor networks. It implements its own bytecode interpreter, built on top of TinyOS. The interpreter provides high-level instructions (such as an atomic message send) which the machine can interpret and execute. Each virtual machine instruction executes in its own TinyOS task.

A somewhat different approach of providing high-level programming abstractions is to view the sensor network as a distributed database, in which sensors produce series of data values and signal processing functions generate abstract data types. The database management engine replaces the virtual machine in that it accepts a query language that allows applications to perform arbitrarily complex monitoring functions. This approach is implemented in the COUGAR sensor network database [5]. A middleware implementation of the same general abstraction is also found in SINA [26], a sensor information networking architecture that abstracts the sensor network into a collection of distributed objects.

Our system is different in that it is geared for environmental tracking applications. To the authors' knowledge, EnviroTrack is the first programming support for sensor networks that explicitly facilitates the coding of tracking applications. Its novel abstractions and underlying mechanisms are well-suited for monitoring targets that move in the physical world. EnviroTrack therefore can have a major impact on application development for sensor networks.

7 Conclusions

This paper introduced the design, implementation, and experimental evaluation of a new distributed programming paradigm and experimental prototype for sensor network applications. The paradigm differs from existing distributed computing models in its central focus on abstracting interactions with a *physical environment* produced by a large array of distributed sensors and actuators. The key advantage of this paradigm lies in its considerable potential to reduce development costs of deeply embedded systems. This reduction comes from offloading from the application developer the details of managing low-level communication, mobility, and group management issues in groups of redundant sensor nodes in tracking applications. Performance results show that in addition to convenient abstractions, efficient implementation is possible in our architecture, in that target tracking is successful at practical target speeds.

This paper might be a first step towards a predictable sensor network "virtual machine" for writing distributed deeply-embedded applications. Such a layer should export reliable behavior and well-defined semantics, implemented on an unreliable, unpredictable, and resource constrained hardware and communication infrastructure. The virtual machine would hide the complexity of sensor network programming from the application developer, making a new more robust and more dynamic realm of sensor network applications attainable to impact future defense, surveillance, habitat monitoring, and disaster management systems.

References

[1] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a sentient computing system. *IEEE Computer*, 34(8):50–56, August 2001.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[3] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[4] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic. An entity maintenance and connection service for sensor networks. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, May 2003.

[5] P. Bonnet, J. Gehrke, and P. Seshardi. Towards sensor database systems. In *2nd International Conference on Mobile Data Management*, pages 3–14, Hong Kong, January 2001.

[6] J. Carter, J. Bennet, and W. Zwaenepoel. Implementation and performance of munin. In *ACM Symposium on Operating Systems Principles*, pages 151–164, October 1991.

[7] A. Cerp, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communication technology. In *ACM Sigcomm Workshop on Data Communication*, San Jose, Costa Rica, April 2001.

[8] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[9] P. Debaty and D. Caswell. Uniform web presence architecture for people, places, and things. *IEEE Personal Communications*, 8(4):46–51, August 2001.

[10] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Mobile networking for smart dust. In *ACM MOBICOM*, Seattle, WA, August 1999.

[11] R. B. et al. On the need for system-level support for ad hoc and sensor networks. *Operating System Review*, 36(2):1–5, April 2002.

[12] Federation of American Scientists Military Analysis Network. http://www.fas.org/man/dod-101/sys/land/row/t72tank.htm.

[13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to network embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[14] J. Heideman, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. *Operating Systems Review*, 35(5):146–159, December 2001.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *ASPLOS*, Cambridge, MA, November 2000.

[16] M. Horton, D. Culler, K. Pister, J. Hill, R. Szewczyk, and A. Woo. Mica: The commercialization of microsensor motes. *Sensors Online*, 19(4), April 2002. http://www.sensorsmag.com/articles/0402/index.htm.

[17] T. Imielinski and S. Goel. Dataspace - querying and monitoring deeply networked collections in physical space. *IEEE Personal Communications*, 7(5):4–9, October 2000.

[18] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *ACM MOBICOM*, Boston, Massachusetts, August 2000.

[19] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *ASPLOS*, San Jose, CA, October 2002.

[20] Magnetic Sensors. http://www.magneticsensors.com/mark_det.html.

[21] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler. Wireless sensor networks for habitat monitoring. In *First ACM International Workshop on Wireless Sensor Networks and Applications*, Atlanta, GA, September 2002.

[22] G. Pei, M. Gerla, and X. Hong. Lanmar: Landmark routing for large scale wireless ad hoc networks with group mobility. In *MobiHoc*, Boston, Massachusetts, August 2000.

[23] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *ACM MOBICOM*, Boston, MA, August 2000.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Sigcomm*, San Diego, CA, August 2001.

[25] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *ACM Symposium on Operating System Principles*, Saint Malo, France, October 1997.

[26] C.-C. Shen, C. Srisathapornphat, and C. Jaikeo. Sensor information networking architecture and applications. *IEEE Personal Communications*, 8(4):52–59, August 2001.

[27] E. Sirer, R. Grimm, A. Gregory, and B. Bershad. 'design and implementation of a distributed virtual machine for networked computers. In *ACM Symposium on Operating System Principles*, pages 202–216, Kiawah Island, SC, December 1999.

[28] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

[29] A. Wood and J. A. Stankovic. Denial of service in sensor networks. *IEEE Computer*, 35(10), October 2002.