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

Abdelzaher T., Blum B., Cao Q., Evans D., George J., George S., He T., Luo L., Son S., Stoleru R., Stankovic J., 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. It contains run-time mechanisms that efficiently abstract groups of sensors by logical objects which maintain aggregate environmental state. Such objects may be logically attached to moving entities in the physical environment, in order to monitor the state of the tracked entity. 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 without overloading the sensor network.

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

1 Introduction

The work reported in this paper is prompted by the increasing importance of distributed sensor networks [17] as a future platform for a growing number of applications such as habitat monitoring [7, 27], intrusion detection [34], defense, and scientific exploration. Advances in hardware miniaturization [11] 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. In this paper, we consider *ad hoc* sensor networks, where a large number of miniature sensor nodes are dropped randomly over an area for monitoring purposes. Several prototypes of such nodes have been recently manufactured, such as Rene, MICA, and Dot Motes [9]. At present, these motes cost around $100 each. It is believed that in the next five years the price will drop to around $1 [18] making massive replication viable. The approach can then be used in many applications such as defense (e.g., to implement low-cost surveillance), disaster relief, and wild habitat monitoring. A key assumption common to these applications is that the nature of the environment prevents the installation of a fixed infrastructure. Hence, the sensor network may operate for large periods of time in isolation without the benefit of external centralized workstations or command centers.

In order to perform its functions, a sensor network operating in isolation must host its own code. An application program is thus distributed among the sensor nodes. It performs activities that are often a direct consequence of particular events that take place in the physical environment. Massive distribution and concurrent interaction with the physical environment call for new programming abstractions. Since environmental tracking is a key application area, tracking-related abstractions are of particular interest. EnviroTrack is a distributed system that implements such abstractions in middleware that runs on sensor devices.

The EnviroTrack library exports a new class of abstractions, called *tracking objects*, that can be dynamically instantiated and logically attached to selected external entities in the physical environment for tracking purposes. Each such object encapsulates the tracked entity's aggregate state. It performs user-defined entity-specific computation and serves as the virtual communication port of the particular tracked physical entity in the application programmer's world. The middleware library off-loads from the programmer the details of inter-object communication, as well as the maintenance of tracking objects and their state. It abstracts away the fact that computation associated with the object is 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

[*]The work reported in this paper was supported in part by the National Science Foundation under grants *[removed for anonymity]* and DARPA under grants *[removed for anonymity]*.

representing it remains the same. The programmer thus interacts with a changing group of sensor nodes through a simple object interface. Simple language support was developed to declare tracking objects.

Using the EnviroTrack interface, programs may include both tracking and static objects. Static objects are conventional and are mentioned for completeness. This model allows sensor network applications to be expressed in a natural and efficient way using abstractions that correspond directly to events in the environment. To implement our abstractions, we provide a run-time system layer that is highly tailored to a sensor network environment and supports dynamic resource tradeoffs.

EnviroTrack has been implemented and tested on a popular sensor network platform based on MICA motes [19]. Our initial implementation of this infrastructure uses compiled NesC [14] programs on TinyOS [17], an operating system for sensor networks. Recent advances in programming support for sensor networks, such as the development of a virtual machine [23], 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. An overview of related work is presented in Section 2. Section 3 defines the tracking problem in more detail, describes our programming system architecture, and elaborates on the main abstractions provided by EnviroTrack. Section 4 illustrates how a sample tracking application can be written in EnviroTrack. Section 5 provides implementation details. Section 6 presents a performance evaluation. The paper concludes with Section 7.

2 Related Work

A growing challenge facing the distributed systems community is to develop programming paradigms and run-time support for the operation of large-scale embedded sensor networks. Sensor networks exhibit several key differences from wired networks and from traditional ad hoc wireless networks. First, they are composed of myriads of small, resource-constrained, unattended sensor nodes, rather than a set of PCs, laptops or PDAs in the possession of their users. Hence, applications must not depend on the correctness or availability of any particular node, but rather should rely on massive redundancy. Second, they are not IP-based. Instead, they feature data-centric addressing and routing protocols usually based on physical geography. Third, sensor node density is much higher than in traditional ad hoc network models, since it is assumed that sensor network nodes are very cheap and can be economically mass-produced and mass-deployed. Finally, and most importantly, sensor networks are seamlessly embedded in a physical environment with which they tightly interact. It is this last difference that motivates the programming paradigm described in this paper.

Classical distributed programming paradigms and middleware such as CORBA [33], group communication [8], remote procedure calls [4], and distributed shared memory [6, 30] 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. This is expected since their main goal is to abstract distributed communication rather than facilitate distributed sensory interactions with an external physical environment. In contrast, this paper offers a new paradigm tailored for sensor networks. As such, the paradigm is centered around "environmentally-inspired" abstractions aimed at simplifying the coding of interactions with the physical world that arise in distributed deeply embedded systems.

Recently, much progress has been made on integrating partial awareness of the physical environment into the computing system. In particular, location-awareness has been investigated at length. Starting with the network layer, location-assisted routing protocols have received much attention such as LAR [22] and DREAM [3]. A real-time version of location-based routing, called RAP, was introduced in [25]. For networks relying on identifier-based routing, scalable location services have been proposed to keep track of locations of identified destinations [24]. System prototypes have been developed in which location was an essential attribute of system objects [16]. The aforementioned protocols are complementary to our research in that they develop location-aware network-layer addressing and routing schemes that can be used for communication between objects in our system. Indeed, we assume that network nodes and routing are location-aware.

The work reported in this paper is more closely related to several recent projects, such as Cricket [28], Sentient Computing [1] and Cooltown [10], that propose high-level paradigms in which an embedded distributed computing system is able to share humans' 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 [28, 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 [20] exports abstractions of physical volumes addressable by their locations. Similarly, directed diffusion [21, 15] and the intentional naming system [2] provide addressing and routing based on data interests [21, 15]. Attributed-based naming is also related to the notion of content-addressable networks [29] 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 [32]. This approach is taken in MagnetOS [12], 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 [23] 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 [31], 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.

3 System Architecture

EnviroTrack abstractions encourage the programmer to think in terms of tracked activities in the physical environment and corresponding tracking actions which need to occur in the computing system. The programmer specifies what constitutes an activity. This specification enables the system to discover and tag those activities and instantiate desired objects in their vicinity such that these activities are tracked. We begin by formally defining the environmental tracking problem.

3.1 Environmental Tracking

Consider an entity e in the physical environment of a sensor network. Let $S_e(t)$ be the set of sensor devices that can *sense* this entity at time t. The semantics of "sense" are application-specific. Hence, the set $S_e(t)$ is formally defined as the set of all the sensor devices for which some boolean function $sense_e()$ of sensory measurements evaluates to true. For example, to detect a fire one might consider a function such as $sense_{fire}() = (temperature > 180)\ and\ (light)$. The set $S_{fire}(t)$ then consists of all nodes for which this function is true at time t.

We define the *exact aggregate state* of the tracked entity, e, as a function $state(S_e(t))$ of the sensory measurements of all devices in set $S_e(t)$. The function $state()$ is application-specific. For example, in the fire-sensing scenario described above, $state()$ might return the average temperature of all sensors in its argument list. In general, the returned data structure could be a vector of different aggregate measurements. Since the membership of $S_e(t)$ may change dynamically at a fast rate, and since message exchange in a distributed system takes non-zero time, it is generally infeasible to maintain the exact aggregate state in real time.

Instead, we define relaxed aggregate state semantics as follows. At any time, T, let us define the *history set* $H_e(T)$ as the union of all sets $S_e(t)$ for $T - L_e \leq t \leq T$, where L_e is called the freshness horizon. L_e defines how fresh the sensed data must be. Intuitively, the history set includes all sensor devices which sensed the tracked entity within the last L_e time units. We now define *approximate aggregate state* as the function $state_e()$ applied to a random *subset* of devices in $H_e(T)$. We call the approximate state *representative* if the subset includes no fewer than than N_e devices, where N_e is the *critical mass constraint*. In other words, approximate state is computed by applying the $state_e()$ aggregation function to at least N_e devices which sensed the tracked event within the last L_e time units. The freshness L_e and the critical mass N_e may be chosen by the domain expert in accordance with application semantics. They represent the QoS parameters of the environmental tracking process. For example, in a fire sensing scenario, it may be enough to consider the output of 5 sensors reporting a fire within a 3 second window. Hence, critical mass $N_{fire} = 5$, and freshness $L_{fire} = 3$ are the QoS constraints of fire tracking.

We define environmental tracking of entity e as the process of maintaining the *approximate aggregate state* of this entity subject to stated freshness and critical mass constraints. EnviroTrack facilitates developing tracking applications by supporting the abstraction of context labels which track physical entities in the environment. Context labels encapsulate and maintain the approximate state of the entities they track. Code can be attached to context labels to perform context-specific computation. In the following, the programming model and abstractions of EnviroTrack are described.

3.2 Programming Model

Humans typically communicate using a set of identifiers, which name objects in the physical world that are defined by specific properties perceivable by the human senses. Such communication is impossible in conventional computing systems due to the lack of appropriate sensory devices that would relay information germane to the definition and identification of the object. Sensor networks offer a unique opportunity to leverage a myriad of available sensing modes (such as temperature, pressure, motion, acceleration, humidity, light, smoke, sound and magnetic field) to develop and communicate perceptions about the physical world. EnviroTrack takes advantage of this opportunity in its 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 example, a mine-locator object sensing a nearby mine can cause its node to detonate itself thereby clearing the threat in a mine-sweeping application. For completeness, EnviroTrack also supports conventional static objects that are not attached to context labels.

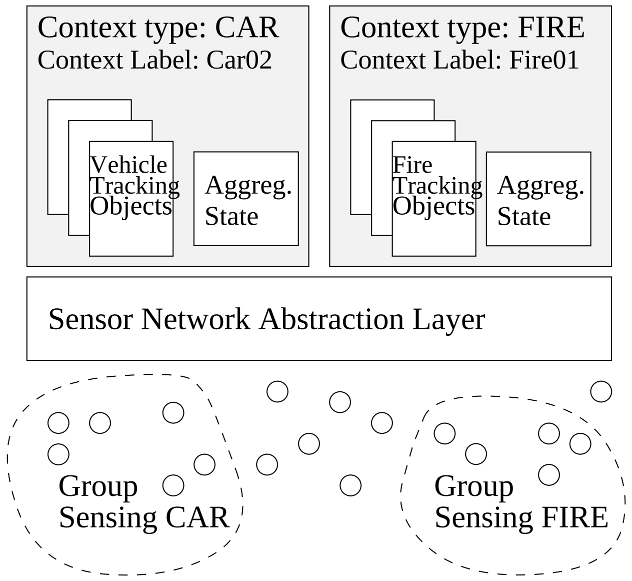

Figure 1. Programming Model

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 the $sense_e()$ function, described in Section 3.1, to define the environmental condition being sensed in this context. This condition is used to watch for the specified pattern in the environment and create a sensor group around the detected target when the pattern occurs. It is also used to maintain the membership of the sensor group around the tracked target when the target moves. Second, the programmer declares what constitutes the approximate environmental state to be encapsulated in the context label. This state is shared by all tracking objects attached. Approximate state is declared by defining the function $state_e()$, described in Section 3.1. EnviroTrack provides a library of the most common distributed aggregation functions to choose from. The underlying infrastructure includes a data collection protocol which collects sensor data (i.e., the arguments for the $state_e()$ function) from sensor group members such that the freshness and critical mass conditions (defined in

Section 3.1) are met for the aggregate state encapsulated in the given context. Note that this protocol is merely for raw sensory data collection and hence is independent of the type of aggregation function performed on the collected data. Finally, the programmer specifies which objects are to be attached to the context label. Attached object code can reference the approximate aggregate state maintained in the enclosing context.

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

3.2.1 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 of 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:

- 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.
- All members of a group at time *t* satisfy the condition $sense_e()$.

Context labels are created dynamically in response to environmental conditions. If a node that senses the activation condition $sense_e()$ of a context has no neighbors detecting the same condition, the node creates a new context label of type e and initializes a new sensor group, becoming the group leader. The group management algorithm is described in more detail in Section 5.2.

3.2.2 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 approximate aggregate state, i.e., they are in accordance with the specified freshness and critical mass requirements.

3.2.3 Approximate Aggregate State

The approximate aggregate state collected in the sensor group of a context label is maintained in a set of variables we call *aggregate state variables*. These variables and their types are statically declared in the definition of the context type. They are replaced at compile time by calls to middleware functions that evaluate their current values. The function $state_e()$ can therefore be thought of as implicitly defined by the aggregate state variables declared for context e.

The numeric value of a state variable is derived by aggregating measurements of individual nodes in the sensor group. Several aggregation functions are provided in the system, such as average, sum, and center of gravity. The definition of a state variable type 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, as well as mechanisms for programming custom aggregation functions.
- 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.

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

In our protocol, nodes that join the group are sent the freshness constraint L_e of all aggregate variables. All members then periodically send to the leader their measurements at a period P_e determined from the freshness constraint L_e. We set $P_e = L_e - d$, 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 *read* operations on aggregate state variables asynchronously with the above protocol. These operations are considered successful if the valid flag was set for the accessed variable. Otherwise, a null flag is set. This situation generally occurs when too few sensors have detected the phenomenon to satisfy the critical mass constraint, i.e., when the "siting" of the phenomenon is not positively confirmed. The programmer is free to handle this situation in any application specific manner, ranging from no action to the invocation of application-specific data analysis functions on the unconfirmed event.

The aforementioned simple algorithm guarantees that a successful read operation on an aggregate variable returns a value that has the following properties at the time the value is returned:

- The value has been computed by the aggregation function whose participants were members of the sensor group, i.e., for whom $sense_e()$ is true.
- The aggregate value was a function of sensor readings that were measured within the freshness threshold L_e of this variable.
- The number of participants who took part in the aggregation was no less than the critical mass N_e.

These conditions satisfy our QoS requirement for approximate aggregate state. Hence, successful environmental tracking can be performed. Note that by selecting weak semantics for approximate aggregate state, we are able to implement aggregations efficiently while providing sufficient guarantees to facilitate writing tracking objects.

4 Language Features and Application Example

To facilitate the use of our middleware, we developed simple language support for declaring context labels and aggregate state variables. A preprocessor uses the stated declaration to emit appropriate code that initializes the middleware. The preprocessor also replaces references to the aggregate state variables by middleware function calls that evaluate them at runtime. An EnviroTrack program consists of a list of context declarations such as the one shown in Figure 2. 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 3.2.3. 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 2 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. EnviroTrack contains a library of such functions for the programmer to choose from. New user-defined functions can be easily added by application developers. 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 $L_e = 1$. No deactivation condition is defined, which defaults to the inverse of the activation condition.

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. It assumes the identity of the pursuer

```
begin context tracker
  activation: magnetic_sensor_reading()
  location : avg (position) confidence=2, freshness=1s

  begin object reporter
    invocation: TIMER(5s)
    report_function() {
      MySend (pursuer, self.label, location);
    }
  end
end context
```

Figure 2. Sample EnviroTrack Code

is known at compile time. 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 pursuer tracks the locations of all reported vehicles, identifying them by their respective context labels.

Observe how the above example encourages the programmer to think in terms of context types. There may be multiple vehicles in the field, in which case the above code will generate multiple instances of the tracker at their respective different locations. Further, even though the vehicles move and the sensor nodes comprising their corresponding objects will change, the context labels will not. This significantly simplifies the programmer's interaction with the varying sensor group tracking each vehicle. A partial grammar of our context definition language is presented in Appendix A.

5 Implementation

In this section, we describe implementation issues in EnviroTrack. Our implementation is built on TinyOS [17], 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 preprocesor translates EnviroTrack declarations such as the one shown in Section 4 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 context label.
- Object naming and directory services: These services maintain lists of currently active objects and their locations.
- The communication and transport services: These services implement remote method invocation.

The aforementioned services are described in the subsections below, respectively.

5.1 The EnviroTrack Preprocessor

The input to the EnviroTrack preprocessor is the context description file, such as the one shown in Section 4. The preprocessor patches a set of NesC program templates using the information gathered from the context description file to produce the target NesC programs. 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.

5.2 Group Management Services

Group management services maintain the notion of context labels. These services have two important goals. First, they maintain the coherence of the context label abstraction. That is, a group of sensors identifying the *same entity* in the environment should produce a *single* context label. This label must persist and remain unique even as the membership of this sensor group changes. Second, these services compute the approximate aggregate state of the context label in accordance with the freshness and critical mass semantics. The reader is reminded that computation of aggregate state in context type e requires that a critical mass N_e of sensors who sense event e respond to the group leader with measurements which satisfy the freshness criterion L_e. This function has already been described in Section 3.2.3. In this section, we focus on the former function, i.e., how context label coherence is maintained. A key design consideration is that group management services must be very lightweight and dynamic. In particular, algorithms which maintain consistent state, or consistent membership views may be too heavyweight. Instead, in our architecture, no single entity has to know the current group membership and no consistent distributed state is assumed.

To maintain context label coherence, at any point in time, nodes sensing the external entity must maintain a single "majority" leader. In our architecture, we allow spurious (i.e., minority) leaders to emerge. These leaders, however, are unlikely to gather critical mass and hence will not affect system behavior.

The leader of a context label sensor group, which tracks an external entity of type e is by definition a member of that group (i.e., $sense_e()$ is true for it). The leader sends periodic heartbeats, which have three purposes. First, they flood the group to inform current members that a leader is alive. Due to the broadcast nature of the wireless medium and because the sensors in a group are physically close, a single message transmission may be enough to flood the group. Second, if needed, they may carry any state that must persist across different timer handler invocations on the leader. This state is null by default, unless explicitly entered by a special EnviroTrack $setState()$ command. This mechanism allows new leaders to continue computations of failed leaders from the last committed state received.[2] Finally, heartbeats may be propagated h hops past the group's perimeter to inform neighboring nodes of the existence of context label e. This informs these nodes that if they subsequently sense the condition $sense_e()$, they should join the present group instead of forming a new context label. The parameter h can be chosen depending on the ratio of the sensing radius and the communication radius. If the communication radius is large enough, h may be zero, since neighboring non-member nodes would hear the leader's broadcast anyway.

Each group member (upon hearing each new heartbeat) sets a *receive timer* which is used to trigger a leadership takeover should the timer expire without receiving a subsequent heartbeat. In our implementation, the receive timer is more than twice longer than the heartbeat period to allow for message loss. This timer serves as a backup to an additional leadership relinquish mechanism which explicitly requests the election of a new leader when the current leader no longer senses event e. In addition, non-members who hear the heartbeat from context e set a *wait timer*. If event e is subsequently sensed before the wait timer expires, the node does not form a new context label, but rather considers itself a new member of the current context label sensor group. The choice of the wait timer depends on how far to maintain memory of nearby events. It should be larger for slower moving events.

Spurious leaders can emerge when heartbeats get lost. Two cases arise. In the first, a member's receive timer can expire which generates a second leader within the context label. In the second case, a non-member's wait timer can expire, which creates a new context label. The first case is relatively harmless. It merely replicates computation of approximate state on two redundant leader nodes. It is likely that one of them will be unable to collect critical mass, and thus show no externally visible action. Otherwise, the context label might appear to produce reports more frequently, but that should not affect a properly written application. To further

[2]In the present prototype, persistent state is not yet implemented. It constitutes a trivial extension to the current algorithm.

reduce redundancy, when a leader hears another leader's heartbeat within the *same* context label group, the leader immediately yields to this leader to prevent confusion and redundant behavior.

The second case (where a non-member creates a spurious context label) is more severe, since the tracked target may then appear replicated to the application. Spurious context labels are reduced by making sure that a node which hears leaders of two *different* context labels of the same type ignore the leader with the smaller *weight*. Each new context label is initially created with a leader weight of zero. Leaders of existing context labels carry a weight equal to the number of messages received by the leader from members to date. This counter increases as sensors report their measurements for computing aggregate state. It is passed during leadership takeover within the context label. Hence, leaders of spurious context labels are likely to be ignored. They are unlikely to collect enough members to satisfy the critical mass constraints. When they receive a heartbeat from a leader of a different context label with the same context type and a larger weight, they delete their context label and become regular members of the other leader's group. The above precautions ensure smooth operation in an unreliable environment with no explicit knowledge of current group membership and no consistent state.

5.3 Object Naming and Directory Services

In order for objects to interact with each other, first they need to be able to find out about other objects. The system provides a way to retrieve references to objects of a given context type. Similar to current research in content distribution networks, we use a hashing function that hashes a type name to some (x, y) coordinate in the sensor network field. The nodes within one hop of that coordinate are responsible for maintaining references to active objects of this type. We will call this set of nodes, the directory object. The directory maintains a mapping from a context label (that provides a unique reference to the object) to a coordinate location for each active object of the type. Thus when a context label first comes alive it computes the hash of its context type and transmits its location information to the directory object for this name. The directory object adds the new context label to the list it is maintaining. Occasional updates from these objects help keep the location information up to date. Hence, the directory object is able to answer queries such as "where are all the fires?". When tracking objects move around, they leave a temporary forwarding pointer behind so that a message sent to the old location is properly forwarded. Once the directory object are updated the forwarding pointers can expire.

5.4 Transport Layer Protocols

Context labels are akin to IP addresses in an Internet environment. The group leader of a context's sensor group oversees all communication with this address. Remote method invocation engages our transport protocol for communication between leaders of the source and destination objects. This protocol is described in more detail in a prior publication [*][3]. In summary, the source object passes the message to the local context leader. This leader identifies the location of the remote context using a directory lookup and communicates the message to the remote context leader, which in turn invokes the destination method. Past leaders act as forwarding routers when the sensor group moves away. Each message contains the current leader of the group, so that future return messages are forwarded as close to the group as possible.

Connections are identified by a <*Context Label, Port Num*> pair. Port IDs are associated with methods of individual objects. The MTP addresses outgoing messages to the last-known leader of the remote context, which is stored locally in a table. Outgoing messages identify the source's current leader in the message header. Upon receiving a message, an endpoint updates its table of last-known leaders with that contained in the header. The more traffic exchanged between the endpoints, the more up-to-date the leader information is.

Leadership information is retained for as long as possible, given limited table sizes. Replacement is done on a least-recently-used basis. This ensures that messages from moderately out-of-date remote senders can be forwarded along a chain of past leaders to the current leader. Observe that while the directory services described in the previous section determine where an object is when it is first contacted, maintaining current information on the location of communicating parties is a responsibility of the transport layer protocol. This optimization obviates additional trips to the directory server for communication on an established connection.

6 Evaluation

In this section we demonstrate the efficacy of EnviroTrack in achieving its primary objective; namely, track physical targets in the environment. 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 prototype.

[3] Removed for anonymity.

6.1 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 [13]. 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 [26]. These sensors operate by detecting 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 to emulate the T-72 per-hop speed calculated above. These values correspond to an emulated speed of 50 km/hr and 33 km/hr, respectively. A single context type was defined, whose declaration is shown in Figure 2. 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 3 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. With a true magnetic sensor, it may be possible to improve the results by estimating proximity using actual magnetic field measurements. In close proximity the sensor can also identify target type from its magnetic signature [26].

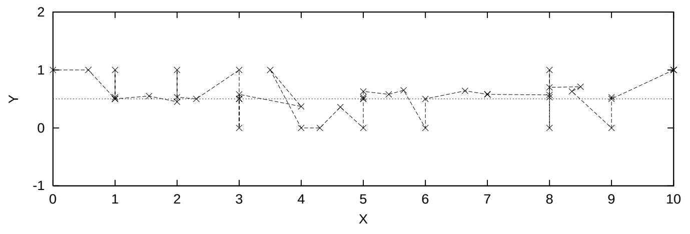

Figure 3. Tracked Tank Trajectory

Figure 4 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 changeover from one member node to another along the target's path. An unsuccessful handover means a new 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 4 shows that a fraction of handovers will fail in this case unless target speed is slow. In the second setting, we ensured that leader heartbeats are propagated one hop past 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 5.2 requires that leader heartbeats be propagated past the sensing radius. The experiment demonstrates the importance of setting the group management parameter h (defined in Section 5.2) and radio transmission radius in a way that does not violate the group management assumptions.

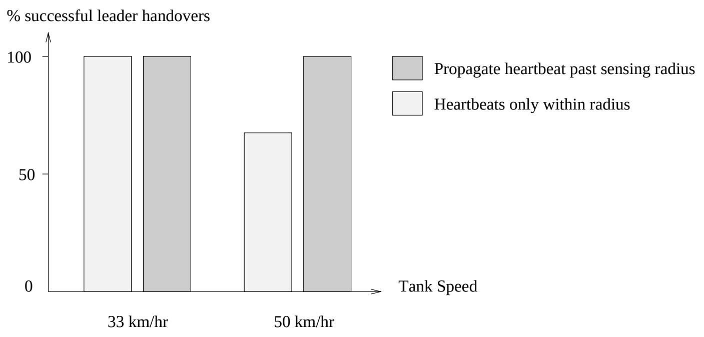

Figure 4. Successful handovers

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 hand-off, 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. The above analysis is only a single case study that did not explore the limitations of the architecture. Recall that the fastest target speed used above is only 0.1 hop/s (or 10 seconds per hop). As we show next, this speed is well below the limits of EnviroTrack. Next, we stress-test the architecture to determine the maximum trackable target speed as a function of various parameter settings of the middleware.

6.2 Testing the Maximum Trackable Speed

The most important parameter which affects the maximum trackable target speed in our architecture is the heartbeat period of the group leader. Best results are achieved when the *receive* and *wait* timers, described in Section 5.2, are set to 2.1 and 4.2 times the leader heartbeat period respectively. The *receive* timer sets the timeout value that initiates leadership takeover upon current leader failure. It is set to slightly more than twice the heartbeat period to allow for two consecutively missed heartbeats before starting the takeover. The *wait* timer is the time (elapsed since the last heard leader heartbeat) that a group member must wait before being allowed to start its own group. To prevent spurious groups from being formed around the same external stimulus during a leadership takeover, the wait timer must be longer than the receive timer.

In our first set of experiments we test the effect of group management timers on the trackable speed. In these experiments, we change the heartbeat period while maintaining the aforementioned timer ratios. For all tests we keep the communication radius at 6 grids and vary the sensor radius from 1 to 2 grids. We then determine the maximum trackable speed for different heartbeat periods and sensor radius settings. 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). This is as opposed to maintaining a single group that migrates along with the target. The maximum trackable speed is the highest target speed at which the single group abstraction is maintained. In other words, it is the highest speed at which context label coherence is ensured. The results of the experiment are shown in Figure 5. 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.

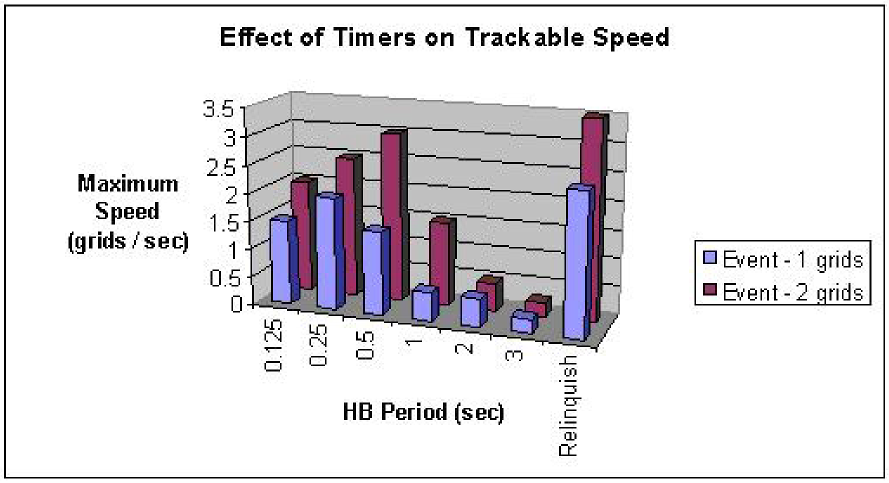

Figure 5. Effect of Timers on Maximum Trackable Speed

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 when the communication radius is fixed (6 grids), events with a larger sensory signature can be tracked at higher speeds. This may seem intuitive, as larger targets should be easier to track. In the next experiment, we further explore the effect of the ratio between the communication radius and size of the target's sensory signature (i.e., sensor detection radius).

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. We see from the figure that we achieve a maximum speed at heartbeat periods of 0.25 and 0.5 for event radii of 1 and 2, respectively. For faster heartbeats (lower periods) our trackable speed diminishes as bandwidth overload restricts the handoff process. Although our data shows that breakdown occurs slightly earlier for larger events, we conjecture that this is just an anomaly resulting from coarse differences between heartbeat periods. Because the number of messages transmitted does not depend on the event size at high speeds (heartbeats and reports are sent at a constant rate and leader updates are sent once by every node as the period of leader updates is longer than the time an event is detected by a node at these speeds), we expect this breakdown point to be around the Heartbeat period of 0.25.

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 5 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 the leadership relinquish optimization previously discussed to improve performance. The results are shown in Figure 6. 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. Because of the leadership relinquish optimization, nodes only communicate when leadership handovers are necessary (a leadership handover is necessary because the old leader stops sensing the target and a new leader must assume the role and announce its status). For a constant speed, when an event is larger, the average time between handovers decreases requiring less bandwidth consuming messages. Fewer handovers result 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. We note that our architecture provides a mechanism for extending the awareness horizon by forwarding leader hearbeats using local flooding restricted to some hop count h. We leave testing of this mechanism and its affect on network congestion and communication to future work.

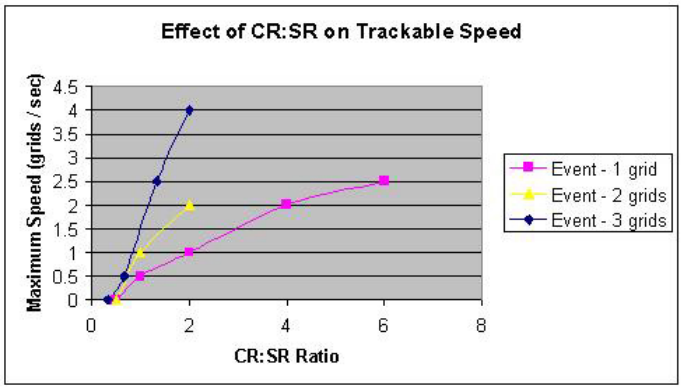

Figure 6. Effect of Sensory Radius on Maximum Trackable Speed

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 off-loading from the application developer the details of managing dynamic groups of redundant sensor nodes in tracking applications, attaching computation to external events in the environment, implementing non-interrupted communication between dynamically changing physical locales defined by such environmental events, representing and maintaining fresh state of mobile external physical entities, and generally building an information infrastructure for tracking environmental conditions. By alleviating these costs, we may have removed one of the fundamental deployment problems facing sensor network applications. Performance results show that in addition to convenient abstractions, efficient data dissimination is possible in our architecture, and target tracking is successful at practical target speeds.

Future work of the authors will involve a more substantial experimental study as well as refinement of the environmental tracking problem such that more precise semantics and failure models are achieved. With such refinements and empirical understanding we hope to build a predictable sensor network "virtual machine" that exports reliable behavior and well-defined semantics, implemented on the unreliable, unpredictable, and resource constrained hardware and communication infrastructure typical of sensor networks. Such a 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 attaintable 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] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and B. A. Woodward. A distance routing effect algorithm for mobility (dream). In *ACM MOBICOM*, pages 76–84, October 1998.

[4] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.
[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] Crossbow Products. http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.
[10] P. Debaty and D. Caswell. Uniform web presence architecture for people, places, and things. *IEEE Personal Communications*, 8(4):46–51, August 2001.
[11] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Mobile networking for smart dust. In *ACM MOBICOM*, Seattle, WA, August 1999.
[12] 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.
[13] Federation of American Scientists Military Analysis Network. http://www.fas.org/man/dod-101/sys/land/row/t72tank.htm.
[14] 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.
[15] 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.
[16] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *IEEE Computer*, 34(8), August 2001.
[17] 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.
[18] T. Hoffman. Smart dust. Computer World, March 2003. http://www.computerworld.com/mobiletopics/mobile/story/0,10801,79572,00.html.
[19] 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.
[20] T. Imielinski and S. Goel. Dataspace - querying and monitoring deeply networked collections in physical space. *IEEE Personal Communications*, 7(5):4–9, October 2000.
[21] 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.
[22] Y.-B. Ko and V. Nitin. Location-aided routing (lar) in mobile ad hoc networks. In *ACM MOBICOM*, pages 66–75, October 1998.
[23] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *ASPLOS*, San Jose, CA, October 2002.
[24] J. Li, J. Jannotti, D. D. Couto, D. Karger, and R. Morris. A scalable location service for goegraphic ad hoc routing. In *ACM MOBICOM*, Boston, Massachusetts, August 2000.
[25] C. Lu, B. Blum, T. Abdelzaher, J. Stankovic, and T. He. Rap: A real-time communication architecture for large-scale wireless sensor networks. In *Real-Time Technology and Applications Symposium*, San Jose, CA, September 2002.
[26] Magnetic Sensors. http://www.magneticsensors.com/mark_det.html.
[27] 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.
[28] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *ACM MOBICOM*, Boston, MA, August 2000.
[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Sigcomm*, San Diego, CA, August 2001.
[30] 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.
[31] C.-C. Shen, C. Srisathapornphat, and C. Jaikeo. Sensor information networking architecture and applications. *IEEE Personal Communications*, 8(4):52–59, August 2001.
[32] 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.
[33] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.
[34] A. Wood and J. A. Stankovic. Denial of service in sensor networks. *IEEE Computer*, 35(10), October 2002.

Appendix A

A partial list of BNF grammar for the new programming language is shown below.

program_statements : program_statements program_statement | program_statement

program_statement : **begin context** Identifier activation_condition context_statements **end context**

context_statements : context_statements context_statement | context_statement

context_statement : aggr_variable_declaration | object_declaration

activation_condition : boolean_expression ';'

aggr_variable_declaration: Identifier ':' aggr_function '(' sensor_names ')' attributes ';'

sensor_names : sensor_names sensor_name | sensor_name

object_declaration : **begin object** Identifier data_declaration function_declarations **end**

function_declarations : one_function | function_declarations one_function

one_function : **invocation** ':' invocation_conditions function_statements

attributes : attributes ',' single_attrib | single_attrib

single_attrib : Identifier '=' Identifier