SwarmOS: A Distributed Computing System for Sensor Networks

(in alphabetic order)
Abdelzaher T., Blum B., Evans D., George J., George S., Huang, C., Nagaraddi P., Sorokin P., Stankovic J.
Department of Computer Science
University of Virginia
Charlottesville, VA 22904

Abstract
The increasing momentum for sensor network deployment as a future distributed computing platform motivates research on appropriate computing paradigms suitable for sensor network applications. Programming abstractions and system support are needed to conveniently manipulate thousands of sensory devices with limited memory and wireless communication capacity, interacting tightly with a physical environment. In this paper, we describe SwarmOS, an object-based distributed system for sensor network programming. The main novel feature of SwarmOS is its seamless integration of objects that live in physical time and space into the computational environment of the application.

1 Introduction
This paper describes a new paradigm and run-time system for distributed computing in sensor networks. Distributed sensor networks (DSN) and swarm systems are gaining importance for an increasing number of applications such as environmental tracking, intrusion detection, defense, and scientific exploration, with many more interesting applications to come. Such systems, for example, may be dropped from the sky on a disaster area to form collaborative teams of programmable nodes that help with rescue operations, or be deployed for surveillance purposes in defense applications. They are characterized by coordination of a very large number of tiny computing elements in a very dynamic physical environment. Application code resides in the network and performs activities that are often a direct consequence of particular events that take place in the physical environment. The environment is thus a main factor which influences the flow and locations of network code execution.

Sensor networks differ from more traditional ad hoc wireless network models in four fundamental respects. First, they are composed of myriads of small unattended sensor nodes, rather than a set of laptops or PDAs in the possession of their users. Second, they are not IP-based, but rather feature data-centric addressing and routing protocols usually based on physical geography. Third, sensor node density (and hence network diameter) are 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. A key challenge facing the distributed systems community is to develop programming paradigms and run-time support for the operation of such large-scale embedded networks. This challenge motivates the work presented in this paper.

Current distributed programming paradigms and middleware such as CORBA [21], group communication (e.g., ISIS [3]), remote procedure calls (RPC [4]), and distributed shared memory (e.g., MUNIN [6]) share in common the fact that their programming abstractions exist in a logical space that does not inherently represent or interact with objects and activities in the physical world. As such, they lay the burden of coding interactions with the physical environment entirely on the application programmer. In contrast, we propose a new object-based programming model and OS infrastructure called SwarmOS that is highly tailored for a sensor network environment to facilitate the tedious task of programming DSN applications.

Sensor networks offer exciting new possibilities in designing operating system support that enables programmers to communicate to the system the intended interactions with the physical environment in a more straightforward and intuitive manner. 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, however, offer a unique opportunity to leverage a myriad of available sensing modes (such as temperature, pressure, motion, vibration, humidity, light, sound, magnetic field, position, velocity, and acceleration) to develop a vocabulary and communicate perceptions which relate to the physical world. SwarmOS supports such a vocabulary, which is called context labels. Programmer defined context labels tag physical objects or activities in the real world. All computation is
attached to one label or another, which establishes the (physical) environmental context in which the computation should execute. We call such computation context-aware.

To give a flavor of application writing with SwarmOS, imagine a set of SwarmOS-enabled nodes lying idle in a DSN. A fire breaks out and a few nodes sense it through their temperature and smoke sensors. These nodes coordinate and create a “Fireman” object. The Fireman object analyzes the fire, and sends an alarm to alert the emergency system. Once the fire is out, the temperature and smoke conditions change - the Fireman object disappears.

The scenario illustrates several forms of new functionality in the network: First, sensor nodes are able to collectively recognize and label an event or activity which occurs in the physical environment. SwarmOS achieves that via context labels, which in this case will create a context called “fire”.

Second, context-specific computation may be deployed at the site of the monitored activity. For example, nearby nodes may perform triangulation to determine the exact location of the fire, or invoke application routines to analyze the chemical composition of the smoke. SwarmOS allows binding of computational objects to neighborhoods of selected physical activities by performing associations between objects and appropriate context labels.

Third, objects may need to collect several sensor readings in a given area and process them using various aggregation functions. SwarmOS facilitates this process by implementing the distributed protocols which perform such aggregation. The programmer merely defines the aggregation function on a logical level as a data type. Variables of that type are maintained by the system and, when accessed, return the result of distributed aggregation. The scope of aggregation is determined by the scope of the labeled (physical) context in which the variable is defined.

Finally, different objects may need to interact. SwarmOS supports efficient directory mechanisms for object registration and location, as well as flexible remote object invocation protocols. The protocols allow communication to occur between objects whose location may migrate in the system due to changes in environmental conditions they track. Hence, the advantages of SwarmOS are:

- The written application code provides a logical description of what activities must be performed and when they should be performed, rather than how various low-level coordination mechanisms should implement them.
- Application execution is context-aware in that code can be associated with a particular physical context (such as a fire) and executed only in the vicinity of such context.
- Code for aggregating and communicating the data sensed from the environment is handled transparently by the SwarmOS.
- The physical environment can be thought of as an active partner which injects, removes, and changes the status of appropriate objects in the programmer’s world to reflect its own current state. The DSN therefore is very dynamic and responsive to rapidly changing environmental conditions.

The rest of this paper is organized as follows. Section 2 elaborates on the main abstraction provided by SwarmOS. Section 3 provides architectural details on the implementation of these programming abstractions. Section 4 illustrates how a sample tracking application can be written in SwarmOS. Section 5 presents a preliminary performance evaluation. Section 6 presents the implementation of our system on a real sensor network hardware prototype, based on motes and TinyOS. The paper concludes with Section 7.

## 2 The Programming Model

Programming in SwarmOS encourages the programmer to think in terms of a reactive model in which specified activities the physical environment cause user-defined actions to occur in the computing system. The programmer specifies only in some abstract way what constitutes an activity. This specification enables the system to monitor and tag those activities. These tags are called context labels. One can imagine the context label as an identifier that logically follows the location of the activity as it moves in the environment. For example, the activity might be a moving vehicle in the sensor network. The corresponding context label identifies a sensor group which is able to sense that vehicle at the present time. While the membership of this group changes as the vehicle moves, the group can be addressed by the same unique label. The programmer is therefore encouraged to forget that indeed a group of sensors is involved in tracking the vehicle. Instead, the programmer thinks of this group as a single mobile object that is logically attached to the vehicle and is able to return data about its current state and location at any time. SwarmOS provides suitable primitives, namely aggregation functions and context state variables, that allow the programmer to write such objects as if they were to execute on a single CPU obliviously to the fact that a distributed group of sensors will be involved in executing object computations and measurements at run time.

The programmer’s world is therefore composed of the set of all such objects, each assigned to track or watch something in the physical environment. Observe that the set of these objects is dynamic as some activities in the environment are born or terminate. The programmer is encouraged to think of the physical environment as an active partner which unilaterally creates new object instances, e.g., as new vehicles enter the field, and destroys old ones when the activities they track cease to exist. Newly created objects can actively update existing objects, e.g., to notify them of newly tracked events.

To summarize, the DSN application is broken down into contexts and objects. Contexts have types, which define categories of application-specific physical conditions in the environment. For example, context type “fire” may be defined as ((temperature > 120) and smoke)
Objects can be triggered automatically by the detection of a particular context in the environment (i.e., when its condition evaluates to true according to current sensor data). To write the application, the programmer defines context types, writes object code, and specifies which object classes are triggered in which contexts.

At run-time, when a set of environmental conditions which match a pre-defined context type are detected in some physical locale by the sensor network, a unique context label is created, and objects associated with that context are activated. For example, a *Fireman* object may be associated with context fire. Assume that this condition is detected in two different spots. Unique context labels are assigned to each instance, e.g., *fire1* and *fire2*. Corresponding instances of the *Fireman* object are created and can be addressed as *fire1.Fireman* and *fire2.Fireman*. Figure 1 depicts a sensor network with several types of objects identifying phenomena of interest sensed at the corresponding locations. Below, we describe some properties that characterize context-aware objects.

![Figure 1. Example Sensor Network](image)

### 2.1 Context Labels and Sensor Groups

Transparency to the programmer, the system maintains for each context label a sensor group, which contains the set of sensors for which the condition labeled by the context type evaluates to true. A sensor node joins this group when its local sensor readings satisfy the context activation condition.

**QUESTION:** HOW DO YOU IMPLEMENT EFFICIENTLY THE MATCHING OF CURRENT SENSOR READINGS TO ALL POSSIBLE CONTEXT TYPES (ACTIVATION CONDITIONS)? CAN YOU BE A PART OF MORE THAN ONE GROUP? It remains in the group until a context deactivation condition is sensed. For example, the deactivation condition for context type “fire” may be (*smoke*) and (*timeout = 10min*), which specifies that no smoke has been observed for the last 10 minutes. Observe that the deactivation conditions are not necessarily the inverse of activation conditions. For example, we omitted any mention of temperature from deactivation conditions of “fire”, since the neighborhood of an extinguished fire may still be very hot. Observe also that the membership of the group can change dynamically over the lifetime of the context label. For example, fire burns out in the original spot while gradually spreading to another. The sensor group associated with a context label maintains two invariants:

- The group is not partitioned. All members of a sensor group can reach each other through other members of the same group.
- All members of a group have satisfied the context activation condition, but have not subsequently satisfied the context deactivation condition.

If a node that senses the activation condition of a context has no neighbours detecting the same context, the node creates a new context label and initializes a new sensor group, becoming the group leader. The leader records the time of group creation. A situation may also occur when a node detecting a particular context realizes it has more than one neighbor detecting the same context type but having different context labels. This occurs if both groups were created disjointly at different locales and have drifted towards each other since. In this case, the node joins the group that was created last. HOW DO YOU MERGE GROUPS?

### 2.2 Context State

Each (physical) context type has state, which represents current environmental measurements. The state is maintained in a set of read-only variables, we call context state variables. These variables and their types are statically declared in the context definition along with the context activation and deactivation conditions. They are visible only within the context in which they are declared, and can be shared by all objects operating in that context.

Context state variables are sensor-based, i.e. their value is derived by aggregating current individual sensor measurements in the sensor group of their context. Several aggregation functions are provided in the system, such as average, median, and center-of-gravity. The type declaration for the context state variable specifies three important pieces of information:

- The aggregation function needed for a context state variable. For example, the location of a magnetic object can be approximately expresses as `centerOfGravity(magneticField)`. The average environmental temperature can be expresses as `average(temperature)`.
- Minimum confidence. This value represents the minimum number of sensor nodes that should be involved in the aggregation for the returned value to be valid. At runtime, the aggregation protocol carried out over the current sensor group members keeps track of the number of its participants. The protocol blocks or returns an error until the number of participants exceeds the minimum confidence of the computed state variable.
- Freshness threshold. This value tells the system how long that variable can "age" before it is considered stale. When
the state variable is referenced while executing the programmer’s code its current value is returned unless the staleness limit has expired, in which case the appropriate function for doing the aggregation is invoked to recalculate a fresh value. The newly computed value is returned. Hence, an access to a context state variable in our programming system is guaranteed to return

From an object-based perspective, context state variables can be thought to be encapsulated by a virtual object called “environment”. The object exports methods to read these variables, but not to write them.

The meaning of the returned aggregate state variable when sensor group membership can change dynamically deserves further clarification. One way to look at it is - taking a snapshot of group membership view of the node referencing the variable and ensuring that all participants of the aggregation have the same view. These strong semantics are impractical in sensor networks where group membership may change at time scales comparable to message transmission times. Our semantics are therefore weaker. In particular, we do not require members to maintain a consistent membership view. Instead, we guarantee that a successful method invocation to read a context 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 where members of the context (sensor group) in which the variable was declared. That is to say, each participant has individually satisfied the context activation condition, but not the deactivation condition.
- The number of participants who took part in the aggregation was no less than the confidence value.
- The aggregate value was a function of sensor reading that were measured within the freshness threshold of this variable.

Context variables with the above semantics greatly facilitate writing context-aware objects. This issue is discussed next.

2.3 Context-Aware Objects

The last part of context declaration is a declaration of attached objects. These objects encapsulate local methods and data. Methods can be invoked externally, or when a context-dependent invocation condition is satisfied. In an object based model, such a context-dependent invocation can be thought of as an invocation by the local virtual “environment” object (i.e., the one defined in the same context). Method activation conditions can be a function of context state variables. For example, to start an acoustic triangulation method as soon as 10 nearby sensors perceive sound, a boolean context state variable of type and(sound) can be defined with a minimum confidence of 10 members.

From the programmer’s perspective, objects are persistent entities. During the object’s lifetime, the set of nodes that constitute its sensor group might change transparently without disrupting the execution of the logical object. Changes in the sensor group membership occur as new sensors join the group when their activation condition becomes true, or leave when it becomes false. Membership changes may also occur due to sensor node failures. Observe that the sensor group is relevant only to the definition of context state variables operated upon by objects in some physical context. Each object, individually, however, is a piece of code that executes on only one node. A hand-off mechanism is invoked to migrate object state when that node quits the sensor group.

To make object state persistent, the state of the CPU for a given object can be saved at programmer-defined points. Local group communication (within the sensor group of the object’s context) replicates any such saved state on all nodes in the group. Hence, the system guarantees that if execution of an object is abruptly interrupted or stopped at any time (due to group membership changes or movement of the group in response to changing environmental inputs), the execution can resume from the last saved state on a different node. SwarmOS is responsible for re-activating on a different node any objects that die in node crashes or because of membership changes, as long as the object’s activation conditions are satisfied. THERE IS A CONSIDERABLE AMOUNT OF HANDWAVING HERE. WE NEED ALGORITHMS AND PROOFS OF SEMANTICS.

Note that the state of the CPU consists of any locals declared by the programmer and the program counter. The aggregate context state variables need not be saved because the programmer gets an up to date value on each reference on an aggregate variable.

3 The Communication Architecture

In this section we describe the communication architecture and protocols that implement group communication within a context’s sensor group, object registration, discovery, and remote object invocation for context-aware objects.

3.1 Group Communication Services

DESCRIBE LEADER ELECTION, HANDOFF, etc.

3.2 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 of retrieving references to objects of a given type (ie: a given object description). Every object description is assigned a name by the programmer and that name is used to retrieve references to all objects that are of the given type and that had registered with the repository. This is done using a hashing function that hashes a name to some x,y co-ordinate in the sensor network field. The nodes at and around that x,y co-ordinate
then take on the responsibility of maintaining references to active objects with this name. We will call this set of nodes, the repository object. It is enough for the repository object to maintain just the location of each active object. Thus when an object first comes alive, it optionally (a object may chose to remain ‘hidden’) does the hash and transmits its location information to the repository object for this name. The repository object adds the new object to the list it is maintaining. Occasional updates from these objects help keep the location information up to date. Note that when objects move around, they leave a ‘trail’ behind. The nodes that are part of the trail, store the object name and the direction in which the object was moving. Thus if somebody sends this object a message to its old position, the trail nodes take care of forwarding the request. The return message would update the sender of the new position of this object.

Note that the whole directory service can be implemented by using our programming paradigm. The repository object is just another object whose context is defined entirely by location. That is to say, the context activation condition for the repository object is “node position is within some ‘r’ distance from the location returned by the hash function” For every object that needs to be exported, we create a repository object description. The repository object contains code to maintain the repository. Thus the moment the sensor network is deployed different repository objects automatically come alive. HOW DO THEY KNOW THE LOCATION AT WHICH THEY SHOULD COME ALIVE WHEN IT IS DEPENDENT ON THE INPUT TO THE HASH FUNCTION? Communicating with these repository objects is no different from communicating with other objects. The only difference is in how a reference to a repository object is obtained, i.e., using a hashing function. Once the reference is obtained, other objects can read the repository object’s data-structure that maintains references to all other objects.

### 3.3 Transport Layer Protocols

Each context, once created, is associated with a team of nodes whose membership changes according to activation and deactivation conditions. Each context instance is individually labeled. Context labels are therefore akin to IP addresses in an Internet environment. The group leader of a context’s sensor group oversees all communication with this address.

Remote object invocation engages our migratory transport protocol (MTP) for communication between leaders of the source and destination objects. 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 delivers it to the destination object. Leaders act as forwarding routers when a team moves away. Each MTP message contains the current leader of the team, so future return messages are forwarded as close to the team as possible.

The MTP provides basic end-to-end transport service between distributed endpoints. Lightweight protocols are desirable in sensor networks because of the relatively high cost of communication. Connections in the MTP keep a minimum of state and impose little packet overhead on messages exchanged.

Connections are identified by a \(<\text{Context Label}, \text{Port Num}>\) pair. Context labels are assumed to be unique, at least probabilistically, in the network. Port IDs are associated with methods of individual objects. The MTP addresses outgoing messages to the last-known leader of the remote team, 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 in the TTP 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 until it finally arrives at the current team leader. Figure ?? shows a remote message addressed to an out-of-date leader, node A. It forwards the message to its own last-known leader, node B, who can then forward it to the current leader. If the current leader replies, the remote endpoint will address subsequent messages directly to it, avoiding the forwarding chain.

The required addressing presents an initialization problem: how to know the context label and leader of a remote team before receiving a message from it? To solve this, the MTP uses the directory services outlined above.

### 4 Application

APPLICATION EXAMPLE. PRESENT HOW TRACKING APPLICATION CODE LOOKS IN SWARM-OS.

### 5 Evaluation

### 6 Related Work

TinyOS [11] is perhaps one of the earliest operating system kernels developed exclusively for sensor nodes. With only 178 bytes of code, 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. While the notion of modular protocol stacks is not new, a great contribution of TinyOS is to implement such a composable framework within the memory and computing constraints of individual sensor nodes.
The need to build context-aware distributed sensing, computing, and actuation systems, which share common perceptions with their users about the physical environment has been most clearly articulated in the sentient computing project [1]. While the full vision of context-aware computing remains a research challenge, 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 [13] and DREAM [2]. A real-time version of location-based routing was introduced in [16]. For networks relying on identifier-based routing, scalable location services have been proposed to keep track of locations of identify destinations [15]. System prototypes have been developed in which location was an essential attribute of system objects [10]. Most such systems, such as Cooltown [7] and Cricket [17], are geared towards a distributed environment of mobile, networked devices that compose a system in which locations of the participants are known and used to provide new services and functionality. In contrast, in sensor networks, locations will be associated with events in the physical environment that may be of interest to network users. This presents additional challenges, since no devices are associated with such events the way networked PDAs or mobile phones may be associated with human users.

Content-addressable networks [18], have been proposed as a paradigm for communication. These are networks in which destinations are addressed by their content attributes, not by their machine identity. For the special case of relatively static content-addressable destinations, directed diffusion has been proposed for a paradigm in which the participants are known and used to provide new services and functionality. The infrastructure maintains an object name space in which names are associated with locales which match certain attribute profiles of the external environment. Flexible rules are applied to determine the matches. The framework is integrated with a capability for in-network processing, which may be initiated at the locations where attribute-based matches occur. Distributed virtual machines have been proposed to provide convenient high-level abstractions to application programmers, while implementing low-level distributed protocols transparently in an efficient manner [20]. This approach is taken in MagnetOS [8], 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.

Mate [14] 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. Code is broken into capsules of 24 single-byte instructions. A send() instruction allows the capsule to be sent to another node as an active message. This provides a mechanism for dissemination of new code into the network via an infection model. The programmer need not worry about coding for each individual sensors, but rather injects code into a single node, and let it diffuse into the network in a virus-like fashion.

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 [19], a sensor information networking architecture that abstracts the sensor network into a collection of distributed objects.

A NICE CONCLUDING PARAGRAPH

7 Conclusions

8 Acknowledgements

The work reported in this paper was supported in part by the National Science Foundation under grants ANI-0105873, CCR-0093144 and CCR-0098269. We would like to thank Xidong Tang for his insightful feedback on the earlier manuscripts of this paper.

References


