Macro-programming Wireless Sensor Networks using *Kairos*

Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan

University of Southern California,
Los Angeles, CA
{gummadi,gnawali,ramesh}@usc.edu
http://enl.usc.edu

Abstract. The literature on programming sensor networks has focused so far on providing higher-level abstractions for expressing *local* node behavior. *Kairos* is a natural next step in sensor network programming in that it allows the programmer to express, in a centralized fashion, the desired *global* behavior of a distributed computation on the entire sensor network. Kairos' compile-time and runtime subsystems expose a small set of programming primitives, while hiding from the programmer the details of distributed-code generation and instantiation, remote data access and management, and inter-node program flow coordination. In this paper, we describe Kairos' programming model, and demonstrate its suitability, through actual implementation, for a variety of distributed programs—both infrastructure services and signal processing tasks—typically encountered in sensor network literature: routing tree construction, localization, and object tracking. Our experimental results suggest that Kairos does not adversely affect the performance or accuracy of distributed programs, while our implementation experiences suggest that it greatly raises the level of abstraction presented to the programmer.

1   Introduction and Motivation

Wireless sensor networks research has, to date, made impressive advances in platforms and software services [1–3]. The utility and practicality of dense sensing using wireless sensor networks has also been demonstrated recently [4–6]. It is now time to consider an essential aspect of sensor network infrastructure—support for *programming* wireless sensor network applications and systems components at a suitably high-level of abstraction. Many of the same reasons that have motivated the re-design of the networking stack for sensor networks (energy-efficiency, different network use models) also motivate a fresh look at programming paradigms for these networks.

Two broad classes of programming models are currently being investigated by the community. One class focuses on providing higher-level abstractions for specifying a node's *local behavior* in a distributed computation. Examples of this approach include the recent work on node-local or region-based abstractions [7, 8]. By contrast, a second class considers programming a sensor network *in the large* (this has sometimes been called *macroprogramming*). One line of research in this class enables a user to declaratively specify a distributed computation over a wireless sensor network, where

the details of the network are largely hidden from the programmer. Examples in this class include TinyDB [9, 10], and Cougar [11].

Kairos' programming model specifies the *global behavior* of a distributed sensornet computation using a *centralized* approach to sensornet programming. Kairos presents an abstraction of a sensor network as a collection of nodes (Section 3) that can all be tasked together simultaneously within a *single* program. The programmer is presented with three constructs: reading and writing variables at nodes, iterating through the one-hop neighbors of a node, and addressing arbitrary nodes. Using only these three simple language constructs, programmers *implicitly* express both distributed data flow and distributed control flow. We argue that these constructs are also natural for expressing computations in sensor networks: intuitively, sensor network algorithms process *named* data generated at individual nodes, often by moving such data to other nodes. Allowing the programmer to express the computation by manipulating *variables* at *nodes* allows us to almost directly use "textbook" algorithms, as we show later in detail in Section 3.2.

Given the single centralized program, Kairos' compile-time and runtime systems construct and help execute a node-specialized version of the compiled program for all nodes within a network. The code generation portion of Kairos is implemented as a language preprocessor add-on to the compiler toolchain of the native language. The compiled binary that is the single-node derivation of the distributed program includes runtime calls to translate remote reads and, sometimes, local writes into network messages. The Kairos runtime library that is present at every node implements these runtime calls, and communicates with remote Kairos instances to manage access to node state. Kairos is *language-independent* in that its constructs can be retrofitted into the toolchains of existing languages.

Kairos (and the ideas behind it) are related to shared-memory based parallel programming models implemented over message passing infrastructures. Kairos is different from these in one important respect. It leverages the observation that most distributed computations in sensor networks will rely on *eventual consistency* of shared node state both for robustness to node and link failure, and for energy efficiency. Kairos' runtime *loosely synchronizes* state across nodes, achieving higher efficiency and greater robustness over alternatives that provide tight distributed program synchronization semantics (such Sequential Consistency, and variants thereof [12]).

We have implemented Kairos as an extension to Python. Due to space constraints of this paper, we describe our implementation of the language extensions and the runtime system in detail in a technical report [13]. On Kairos, we have implemented three distributed computations that exemplify system services and signal processing tasks encountered in current sensor networks: constructing a shortest path routing tree, localizing a given set of nodes [2], and vehicle tracking [14]. We exhibit each of them in detail in Section 3 to illustrate Kairos' expressibility. We then demonstrate through extensive experimentation (Section 4) that Kairos' level of abstraction does not sacrifice *performance*, yet enables *compact* and *flexible* realizations of these fairly sophisticated algorithms. For example, in both the localization and vehicle tracking experiments, we found that the performance (convergence time, and network message traffic) and accuracy of Kairos are within 2x of the reported performance of explicitly distributed
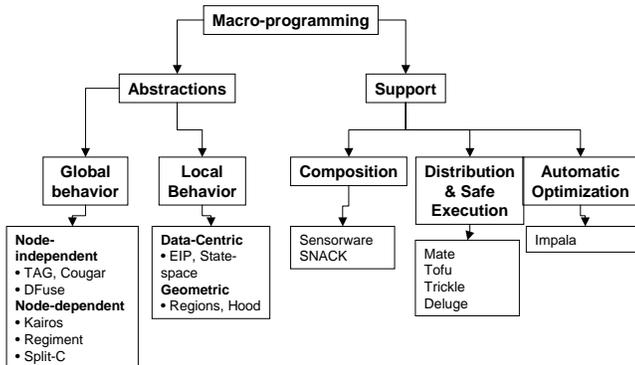
Fig. 1. Taxonomy of Programming Models for Sensor Networks

original versions, while the Kairos versions of the programs are more succinct and, we believe, are easier to write.

2 Related Work

In this section, we give a brief taxonomy (Figure 1) of sensornet programming and place our work in the context of other existing work in the area. The term "sensornet programming" seems to refer to two broad classes of work that we categorize as *programming abstractions* and *programming support*. The former class is focused on providing programmers with abstractions of sensors and sensor data. The latter is focused on providing additional runtime mechanisms that simplify program execution. Examples of such mechanisms include safe code execution, or reliable code distribution.

We now consider the research on sensor network programming abstractions. Broadly speaking, this research can be sub-divided into two sub-classes: one sub-class focuses on providing the programmer abstractions that simplify the task of specifying the node *local behavior* of a distributed computation, while the second enables programmers to express the *global behavior* of the distributed computation.

In the former sub-class, three different types of programming abstractions have been explored. For example, Liu *et al.* [15] and Cheong *et al.* [16] have considered node group abstractions that permit programmers to express communication within groups sharing some common group state. Data-centric mechanisms are used to efficiently implement these abstractions. By contrast, Mainland *et al.* [8] and Whitehouse *et al.* [7] show that topologically defined group abstractions ("neighborhoods" and "regions" respectively) are capable of expressing a number of local behaviors powerfully. Finally, the work on EIP [17] provides abstractions for physical objects in the environment, enabling programmers to express tracking applications.

Kairos falls into the sub-class focused on providing abstractions for expressing the global behavior of distributed computations. One line of research in this sub-class pro-

vides *node-independent* abstractions—these programming systems do not contain explicit abstractions for nodes, but rather express a distributed computation in a network-independent way. Thus, the work on SQL-like expressive but Turing-incomplete query systems (*e.g.,* TinyDB [10, 9] and Cougar [11]), falls into this class. Another body of work provides support for expressing computations over logical topologies [18, 19] or task graphs [20] which are then dynamically mapped to a network instance. This represents a plausible alternative to macroprogramming sensor networks. However, exporting the network topology as an abstraction can impose some rigidity in the programming model. It can also add complexity to maintaining the mapping between the logical and the physical topology when nodes fail.

Complementary to these approaches, *node-dependent* abstractions allow a programmer to express the global behavior of a distributed computation in terms of nodes and node state. Kairos, as we shall discuss later, falls into this class. As we show, these abstractions are natural for expressing a variety of distributed computations. The only other piece of work in this area is Regiment [21], a recent work. While Kairos focuses on a narrow set of flexible language-agnostic abstractions, Regiment focuses on exploring how *functional programming* paradigms might be applied to programming sensor networks in the large, while Split-C [22] provides "split" local-global address spaces to ease parallel programming that Kairos also provides through the remote variable access facility, but confines itself to the "C" language that lacks a rich object-oriented data model and a language-level concurrency model . Therefore, the fundamental concepts in these two works are language-specific.

Finally, quite complementary to the work on programming abstractions is the large body of literature devoted to systems in support of network programming. Such systems enable high-level composition of sensor network applications (Sensorware [23] and SNACK [24]), efficient distribution of code (Deluge [25]), support for sandboxed application execution (Maté [26]), and techniques for automatic performance adaptation (Impala [27]).

3   Kairos Programming Model

In this section, we describe the Kairos abstractions and discuss their expressibility and flexibility using three canonical sensor network distributed applications: routing tree construction, ad-hoc localization, and vehicle tracking.

3.1   Kairos Abstractions and Programming Primitives

Kairos is a simple set of extensions to a programming language that allows programmers to express the global behavior of a distributed computation. Kairos extends the programming language by providing three simple abstractions.

The first of these is the *node* abstraction. Programmers explicitly manipulate nodes and lists of nodes. Nodes are logically *named* using integer identifiers. The logical naming of nodes does *not* correspond to a topological structure. Thus, at the time of program composition, Kairos does not require programmers to specify a network topology. In Kairos, the `node` datatype exports operators like equality, ordering (based on node

name), and type testing. In addition, Kairos provides a `node_list` iterator data type for manipulating node sets.

The second abstraction that Kairos provides is the list of *one-hop neighbors* of a node. Syntactically, the programmer calls a `get_neighbors()` function. The Kairos runtime returns the current list of the node's radio neighbors. Given the broadcast nature of wireless communication, this is a natural abstraction for sensor network programming (and is similar to *regions* [8], and *hoods* [7]). Programmers are exposed to the underlying network topology using this abstraction. A Kairos program typically is specified in terms of operations on the neighbor list; it may construct more complex topological structures by iterating on these neighbors.

The third abstraction that Kairos provides is *remote data access*, namely the ability to read from variables at named nodes. Syntactically, the programmer uses a `variable@node` notation to do this. Kairos itself does not impose any restrictions on which remote variables may be read where and when. However, Kairos' compiler extensions respect the scoping, lifetime, and access rules of variables imposed by the language it is extending. Of course, variables of types with node-local meaning (*e.g.,* file descriptors, and memory pointers) cannot be meaningfully accessed remotely.

Node Synchronization: Kairos' remote access facility effectively provides a shared-memory abstraction across nodes. The key challenge (and a potential source of inefficiency) in Kairos is the messaging cost of synchronizing node state. One might expect that nodes would need to synchronize their state with other nodes (update variable values at other nodes that have cached copies of those variables, or coordinate writes to a variable) often. In Kairos, only a node may write to its variable, thus mutually exclusive access to remote variables is not required; thereby, we also eliminate typically subtle distributed programming bugs arising from managing concurrent writes.

Kairos leverages another property of distributed algorithms for sensor networks in order to achieve low overhead. We argue that, for fairly fundamental reasons, distributed algorithms will rely on a property we call *eventual consistency*: individual intermediate node states are not guaranteed to be consistent, but, in the absence of failure, the computation eventually converges. This notion of eventual consistency is loosely molded on similar ideas previously proposed in well-known systems such as Bayou [28]. The reason for this, is, of course, that sensor network algorithms need to be highly robust to node and link failures, and many of the proposed algorithms for sensor networks use soft-state techniques that essentially permit only eventual consistency.

Thus, Kairos is designed under the assumption that *loose synchrony* of node state suffices for sensor network applications. Loose synchrony means that a read from a client to a remote object blocks *only* until the referenced object is initialized and available at the remote node and *not* on every read to the remote variable. This allows nodes to synchronize changed variables in a lazy manner, thereby reducing communication overhead. However, a reader might be reading a stale value of a variable, but because of the way distributed applications are designed for sensor networks, the nodes eventually converge to the right state. Where this form of consistency is inadequate, we provide a tighter consistency model, as described at the end of this section.

The Mechanics of Kairos Programming: Before we discuss examples of programming in Kairos, we discuss the mechanics of programming and program execution (Fig-
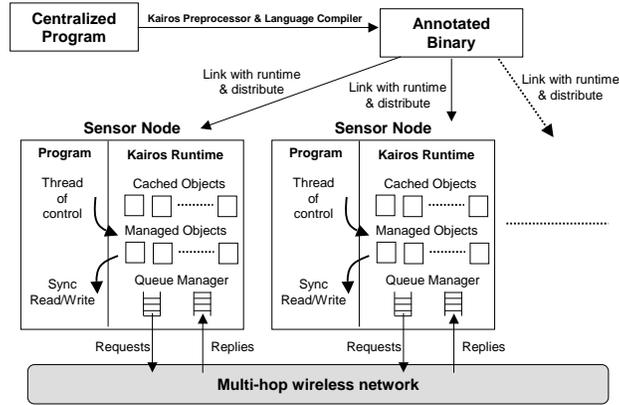
Fig. 2. Kairos Programming Architecture

ure 2). As we have said before, the distinguishing feature of Kairos is that programmers write a single *centralized* version of the distributed computation in a programming language of their choice. This language, we shall assume, has been extended to incorporate the Kairos abstractions. For ease of exposition, assume that a programmer has written a centralized program **P** that expresses a distributed computation; in the rest of this section, we discuss the transformations on **P** performed by Kairos.

Kairos' abstractions are first processed using a *preprocessor* which resides as an extension to the language compiler. Thus, **P** is first pre-processed to generate annotated source code, which is then compiled into a binary $\mathbf{P_b}$ using the native language compiler. While **P** represents a global specification of the distributed computation, $\mathbf{P_b}$ is a node-specific version that contains code for what a single node does at any time, and what data, both remote and local, it manipulates.

In generating $\mathbf{P_b}$, the Kairos preprocessor identifies and translates references to remote data into calls to the Kairos *runtime*. $\mathbf{P_b}$ is linked to the Kairos runtime and can be distributed to all nodes in the sensor network through some form of code distribution and node re-programming facility [29, 25]. When a copy is instantiated and run on each sensor node, the Kairos runtime exports and manages program variables that are owned by the current node but are referenced by remote nodes; these objects are called *managed objects* in Figure 2. In addition, it also caches copies of managed objects owned by remote nodes in its *cached objects* pool. Accesses to both sets of objects are managed through queues as asynchronous request/reply messages that are carried over a potentially multihop radio network.

The user program that runs on a sensor node calls *synchronously* into Kairos runtime for reading remote objects, as well as for accessing local managed objects. These synchronous calls are automatically generated by the preprocessor. The runtime accesses these cached and managed objects on behalf of the program after suspending the calling thread. The runtime uses additional background threads to manage object queues,

```
1:  void buildtree(node root)
2:    node parent, self;
3:    unsigned short dist_from_root;
4:    node_list neighboring_nodes, full_node_set;
5:    unsigned int sleep_interval=1000;
      //Initialization
6:    full_node_set=get_available_nodes();
7:    for (node temp=get_first(full_node_set); temp!=NULL; temp=get_next(full_node_set))
8:      self=get_local_node_id();
9:      if (temp==root)
10:       dist_from_root=0; parent=self;
11:     else dist_from_root=INF;
12:     neighboring_nodes=create_node_list(get_neighbors(temp));
13:   full_node_set=get_available_nodes();
14:   for (node iter1=get_first(full_node_set); iter1!=NULL; iter1=get_next(full_node_set))
15:     for(;;)  //Event Loop
16:       sleep(sleep_interval);
17:       for (node iter2=get_first(neighboring_nodes); iter2!=NULL; iter2=get_next(neighboring_nodes))
18:         if (dist_from_root@iter2+1<dist_from_root)
19:           dist_from_root=dist_from_root@iter2+1;
20:           parent=iter2;
```

Fig. 3. Procedural Code for Building a Shortest-path Routing Tree

but this aspect is transparent to the application, and the application is only aware of the usual language threading model.

3.2 Examples of Programming with Kairos

We now illustrate Kairos' expressibility and flexibility by describing how Kairos may be used to program three different distributed computations that have been proposed for sensor networks: routing tree construction, localization, and vehicle tracking.

Routing Tree Construction In Figure 3, we illustrate a *complete* Kairos program for building a routing tree with a given root node. We have implemented this algorithm, and evaluate its performance in Section 4. Note that our program implements shortest-path routing, rather than selecting paths based on link-quality metrics [30]: we have experimented with the latter as well, as we describe below.

The code shown in Figure 3 captures the essential functionality involved in constructing a routing tree while maintaining brevity and clarity. It shows how a centralized Kairos task looks, and illustrates how the Kairos primitives are used to express such a task. Program variable dist_from_root is the only variable that needs to be remotely accessed in lines 18-19, and is therefore a *managed object* at a source node and a *cached object* at the one-hop neighbors of the source node that programmatically read this variable. The program also shows how the node and node_list datatypes and their API's are used. get_available_nodes() in lines 6 and 13 instructs the Kairos preprocessor to include the enclosed code for each iterated node; it also provides an iterator handle that can be used for addressing nodes from the iterator's perspective, as shown in line 12. Finally, the program shows how the get_neighbors() function is used in line 12 to acquire the one-hop neighbor list at every node.

The event loop between lines 15-20 that runs at all nodes eventually picks a shortest path from a node to the root node. Our implementation results show that the path monotonically converges to the optimal path, thereby demonstrating progressive correctness. Furthermore, the path found is stable and does not change unless there are transient or permanent link failures that cause nodes to be intermittently unreachable.

This event loop illustrates how Kairos leverages eventual consistency. The access to the remote variable `dist_from_root` need not be synchronized at every step of the iteration; the reader can use the current cached copy, and use a lazy update mechanism to avoid overhead. As we shall see in Section 4, the convergence performance and the message overhead of loose synchrony in real-world experiments is reasonable. We also tried metrics other than shortest hop count (such as fixing parents according to available bandwidth or loss rates, a common technique used in real-world routing systems [3]), and we found that the general principle of eventual consistency and loose synchrony can be applied to such scenarios as well.

Let us examine Figure 3 for the flexibility programming to the Kairos model affords. If we want to change the behavior of the program to have the tree construction algorithm commence at a pre-set time that is programmed into a base station node with id 0, we could add a single line before the start of the `for(){}` loop at line 7: `sleep(starting_time@0-get_current_time())`. The runtime would then automatically fetch the `starting_time` value from node 0.

Distributed Localization using Multi-lateration Figure 4 gives a complete distributed program for collaboratively fixing the locations of nodes with unknown coordinates. The basic algorithm was developed by Savvides *et al.* [2]. Our goal in implementing this algorithm in Kairos was to demonstrate that Kairos is flexible and powerful enough to program a relatively sophisticated distributed computation. We also wanted to explore how difficult it would be to program a "textbook" algorithm in Kairos, and compare the relative performance of Kairos with the reported original version (Section 4).

The goal of the "cooperative multi-lateration" algorithm is to compute the locations of all unknown nodes in a connected meshed wireless graph given ranging measurements between one-hop neighboring nodes and a small set of beacon nodes that already know their position. Sometimes, it may happen that there are not enough beacon nodes in the one-hop vicinity of an unknown node for it to mathematically laterize its location. The basic idea is to iteratively search for enough beacons and unknown nodes in the network graph so that, taken together, there are enough measurements and known co-ordinates to successfully deduce the locations of all unknown nodes in the sub-graph.

Figure 4 shows the complete code for the cooperative multi-lateration algorithm.[1] The code localizes non-beacon nodes by progressively expanding the subgraph, (`subgraph_to_localize`), considered at a given node with next-hop neighbors of unlocalized leaf vertexes (`unlocalized_leaves`), and is an implementation of Savvides' algorithm [2]. The process continues until either all nodes in the graph are considered (lines 20-25) and the graph is deemed unlocalizable, or until the initiator localizes itself (using the auxiliary function `subgraph_check()`) after acquiring a sufficient number of beacon nodes. This program once again illustrates eventual consistency because the variable `localized@node` is a monotonic boolean, and eventually attains its correct asymptotic value when enclosed in an event loop. We also found an interesting evidence to the value of Kairos' centralized global program specification approach—we encountered a subtle logical (corner-case recursion) bug in

---

[1] Of course, we have not included the low-level code that actually computes the range estimates using ultrasound beacons. Our code snippet assumes the existence of node-local OS/library support for this purpose.

```
1: void CooperativeMultilateration()

2:  boolean localized=false, not_localizable=false, is_beacon=GPS_available();
3:  node self=get_local_node_id();
4:  graph subgraph_to_localize=NULL;

5:  node_list full_node_set=get_available_nodes();
6:  for (node iter=get_first(full_node_set); iter!=NULL; iter=get_next(full_node_set)))
       //At each node, start building a localization graph
7:    participating_nodes=create_graph(iter);
8:    node_list neighboring_nodes=get_neighbors(iter);
9:    while ((!localized || !is_beacon) && !not_localizable)
10:     for (node temp=get_first(neighboring_nodes); temp!=NULL; temp=get_next(neighboring_nodes))
          //Extend the subgraph with neighboring nodes
11:       extend_graph(subgraph_to_localize, temp, localized@temp||is_beacon@temp?beacon:unknown);
        //See if we can localize the currently available subgraph
12:     if (graph newly_localized_g=subgraph_check(subgraph_to_localize))
13:       node_list newly_localized_l=get_vertices(newly_localized_g);
14:       for (node temp=get_first(newly_localized_l); temp!=NULL; temp=get_next(newly_localized_l))
15:         if (temp==iter) localized=true;
16:           continue;
        //If not, add nodes adjacent to the leaves of the accumulated subgraph and try again
17:     node_list unlocalized_leaves;
18:     unlocalized_leaves=get_leaves(subgraph_to_localize);
19:     boolean is_extended=false;
20:     for (node temp=get_first(unlocalized_leaves); temp!=NULL; temp=get_next(unlocalized_leaves))
21:       node_list next_hop_l=get_neighbors(temp);
22:       for (node temp1=get_first(next_hop_l); temp1!=NULL; temp1=get_next(next_hop_l))
23:         extend_graph(subgraph_to_localize, temp1, localized@temp1||is_beacon@temp1?beacon:unknown);
24:         is_extended=true;
25:     if (!is_extended) not_localizable=true;
```

Fig. 4. Procedural Code for Localizing Sensor Nodes

the original algorithm described in [2] in a local (*i.e.,* bottom-up, node-specific) manner, that became apparent in Kairos.

Vehicle Tracking For our final example, we consider a qualitatively different application: tracking moving vehicles in a sensor field. The program in Figure 5 is a straightforward translation of the algorithm described in [14]. This algorithm uses probabilistic techniques to maintain belief states at nodes about the current location of a vehicle in a sensor field. Lines 14-16 correspond to step 1 of the algorithm given in [14, p. 7] where nodes diffuse their beliefs about the vehicle location. Lines 17-21 compute the probability of the observation z_{t+1} at every grid location given vehicle location x_{t+1} at time $t + 1$ (step 2 of the algorithm) using the latest sensing sample and vehicle dynamics. Lines 23-25 compute the overall posteriori probability of the vehicle position on the rectangular grid after incorporating the latest posteriori probability (step 3 of the algorithm). Finally, lines 26-40 compute the information utilities, I_k's, at all one-hop neighboring nodes k for every node, and pick that $k = \text{argmax}\,I_k$ that maximizes this measure (steps 4 and 5). This node becomes the new "master" node: *i.e.,* it executes the steps above for the next epoch, using data from all other nodes in the process.

This program illustrates an important direction of future work in Kairos. In this algorithm, the latest values of $p(z_{t+1}|\overline{x_{t+1}})[x][y]$@neighbors must be used in line 33 at the master because these $p(.)[x][y]$'s are computed at each sensor node using the latest vehicle observation sample. With our loose synchronization model, we cannot insure that the master uses these latest values computed at the remote sensor nodes because stale cached values may be returned instead by the master Kairos runtime, thereby adversely impacting the accuracy and convergence time of the tracking application. There are two possible solutions to this. One, which we have implemented currently in Kairos, is to provide a slightly tighter synchronization model that we call *loop-level synchrony*,

```
1: void track_vehicle()
2:   boolean master=true;
3:   float z_{t+1}, normalizing_const;
4:   float p(x_t|\overline{z_t})[MAX_X][MAX_Y], p(x_{t+1}|\overline{z_t})[MAX_X][MAX_Y],
            p(z_{t+1}|\overline{x_{t+1}})[MAX_X][MAX_Y], p(x_{t+1},z^k_{t+1}|\overline{z_t})[MAX_X][MAX_Y], p(z^k_{t+1}|\overline{z_t}), p(x_{t+1}|\overline{z_{t+1}})[MAX_X][MAX_Y];
5:   float max_I_k=I_k; node argmax_I_k, self=get_local_node_id();
6:   node_list full_node_set=get_available_nodes();
7:   for (node iter=get_first(full_node_set); iter!=NULL; iter=get_next(full_node_set))
8:     for (int x=0; x<MAX_X; x++)
9:       for (int y=0; y<MAX_Y; y++)
10:          p(x_t|\overline{z_t})[x][y]= \frac{1}{MAX\_X \times MAX\_Y} ;
11:   for(;;)
12:     sleep();
13:     if (master)
14:       for (int x=0; x<MAX_X; x++)
15:         for (int y=0; y<MAX_Y; y++)
16:            p(x_{t+1}|\overline{z_t})[x][y]= \sum_{0 \le x' < MAX\_X} \sum_{0 \le y' < MAX\_Y} \frac{\delta(\sqrt{x'^2+y'^2} - \sqrt{x^2+y^2} - v)p(x_t|\overline{z_t})}{\delta(\sqrt{x'^2+y'^2} - \sqrt{x^2+y^2} - v)} ;
17:       z_{t+1}=sense_z();
18:       normalizing_const=0;
19:       for (int x=0; x<MAX_X; x++)
20:         for (int y=0; y<MAX_Y; y++)
21:            p(z_{t+1}|x_{t+1})[x][y]= \frac{r}{\delta_a} \left[ \Phi\left(\frac{a_{hi}-rz}{r\sigma}\right) - \Phi\left(\frac{a_{lo}-rz}{r\sigma}\right) \right];
22:            normalizing_const+=p(z_{t+1}|x_{t+1})[x][y] \cdot p(x_{t+1}|\overline{z_t})[x][y];
23:       for (int x=0; x<MAX_X; x++)
24:         for (int y=0; y<MAX_Y; y++)
25:            p(x_{t+1}|\overline{z_{t+1}})[x][y]= \frac{p(z_{t+1}|x_{t+1})[x][y] \cdot p(x_{t+1}|\overline{z_t})[x][y]}{normalizing\_const} ;

26:     node_list neighboring_nodes=get_neighbors(iter);
27:     append_to_list(neighboring_nodes, self);
28:     max_I_k=-\infty; argmax_I_k=self;
29:     for (node temp=get_first(neighboring_nodes); temp!=NULL; temp=get_next(neighboring_nodes))
30:       p(z^k_{t+1}|\overline{z_t})=0;
31:       for (int x=0; x<MAX_X; x++)
32:         for (int y=0; y<MAX_Y; y++)
33:            p(x_{t+1},z^k_{t+1}|\overline{z_t})[x][y]=p(z_{t+1}|\overline{x_{t+1}})[x][y]@temp \cdot p(x_{t+1}|\overline{z_t})[x][y];
34:            p(z^k_{t+1}|\overline{z_t})+=p(x_{t+1},z^k_{t+1}|\overline{z_t})[x][y];
35:       for (int x=0; x<MAX_X; x++)
36:         for (int y=0; y<MAX_Y; y++)
37:            I_k += \log \left[ \frac{p(x_{t+1},z^k_{t+1}|\overline{z_t})[x][y]}{p(x_{t+1}|\overline{z_t})[x][y]p(z^k_{t+1}|\overline{z_t})} \right] \cdot p(x_{t+1},z^k_{t+1}|\overline{z_t})[x][y];
38:       if (max_I_k<I_k) argmax_I_k=temp;
39:     if (argmax_I_k!=self) master=false;
40:     master@argmax_I_k=true;
```

Fig. 5. Procedural Code for Vehicle Tracking

where variables are synchronized at the beginning of an event loop (at line 11 of every iteration). A more general direction, which we have left for future work is to explore *temporal data abstractions*. These would allow programmers to express which samples of the time series $p(.)[x][y]$ from remote nodes are of interest, while possibly allowing Kairos to preserve loose synchrony.

4 Kairos Evaluation

We have implemented the programming primitives discussed in the previous section in Python using its embedding and extendability API's [31], and have experimented with the three distributed algorithms described therein. More discussion about our implementation and evaluation can be found in [13]. Our testbed is a hybrid network of ground nodes and nodes mounted on a ceiling array. The 16 ground nodes are Stargates [32] that each run Kairos. In this setup, Kairos uses Emstar [33] to implement end-to-end reliable routing and topology management. Emstar, in turn, uses a Mica2 mote [34] mounted on the Stargate node (the leftmost picture in Figure 6 shows a single Stargate+Mica2 node) as the underlying network interface controller (NIC) to achieve

realistic multihop wireless behavior. These Stargates were deployed in a small area (middle picture in Figure 6), making all the nodes reachable from any other node in a single physical hop (we created logical multihops over this set in the experiments below). The motes run TinyOS [35], but with S-MAC [36] as the MAC layer.

There is also an 8-node array of Mica2dots [37] mounted on a ceiling (rightmost picture in Figure 6), and connected through a multiport serial controller to a standard PC that runs 8 Emstar processes. Each Emstar process controls a single Mica2dot and is attached to a Kairos process that also runs on the host PC. This arrangement allows us to extend the size of the evaluated network while still maintaining some measure of realism in wireless communication. The ceiling Mica2dots and ground Mica2s require physical multihopping for inter-node communication. The Mica2dot portion of the network also uses physical multihopping for inter-node communication.
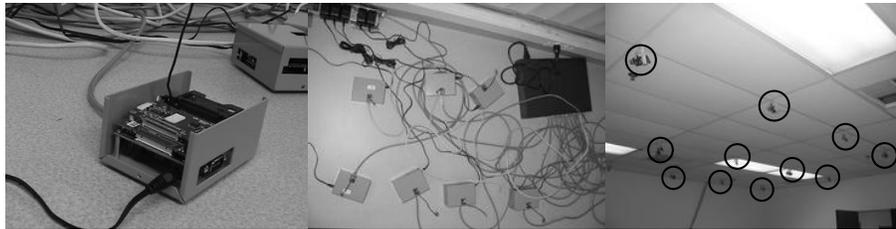
Fig. 6. Stargate with Mica2 as a NIC (left), Stargate Array (middle), and Ceiling Mica2dot Array (right)

To conduct experiments with a variety of controlled topologies, we wrote a topology manager in Emstar that enables us to specify neighbors for a given node and blacklist/whitelist a given neighbor. Dynamic topologies were simulated by blacklisting/whitelisting neighbors while the experiment was in progress. The end-to-end reliable routing module keeps track of all the outgoing packets (on the source node) and periodically retransmits the packets until an acknowledgment is received from the destination. Hop-by-hop retransmission by S-MAC is complementary and used as a performance enhancement.

Routing Tree Performance: We implemented the routing tree described in Section 3.2 in Kairos, and measured its performance. For comparison purposes, we also implemented One Phase Pull (OPP) [38] routing directly in Emstar. OPP forms the baseline case because it is the latest proposed refinement for directed-diffusion that is designed to be traffic-efficient by eliminating exploratory data messages: the routing tree is formed purely based on interest–requests (interest messages in directed diffusion) that are flooded to the network and responses (data) are routed along the gradients setup by the interest. To enable a fair comparison of the Kairos routing tree with OPP, we also implemented reliable routing for OPP.

We varied the number of nodes in our network, and measured the time it takes for the routing tree in each case to stabilize (convergence time), and the overhead incurred in doing so. In the case of OPP, the resulting routing tree may not always be the shortest
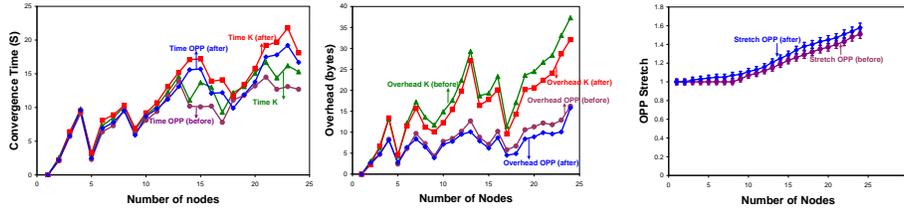
Fig. 7. Convergence Time (left), Overhead (middle), and OPP Stretch (right) for the Routing Tree Program

path routing tree (directed diffusion does not require that), while Kairos always builds a correct shortest path routing tree. So we additionally measure the "stretch" (the averaged node deviation from the shortest path tree) of the resulting OPP tree with respect to the Kairos shortest path tree. Thus, this experiment serves as a benchmark for efficiency and correctness metrics for Kairos' eventual consistency model.

We evaluated two scenarios: first to build a routing tree from scratch on a quiescent network, and second to study the dynamic performance when some links are deleted after the tree is constructed. Figure 7 shows the convergence time ("K" is for Kairos, and "before" and "after" denote the two scenarios before and after link failures), overhead, and stretch plots for OPP and Kairos averaged across multiple runs; for stretch, we also plot the OPP standard deviation. It can be seen that Kairos always generates a better quality routing tree than OPP (OPP stretch is higher, especially as the network size increases) without incurring too much higher convergence time ($\sim$30%) and byte overhead costs ($\sim$2x) than OPP.

Localization: We have implemented the collaborative multilateration algorithm described in Section 3.2. Since we did not have the actual sensors (ultrasound and good radio signal strength measurement) for ToA (Time of Arrival) ranging, we hard-coded the pairwise distances obtained from a simulation as variables in the Kairos program instead of acquiring them physically. We believe this is an acceptable artifact that does not compromise the results below. We perturbed the pairwise distances with white Gaussian noise (standard deviation 20mm to match experiments in [2]) to reflect the realistic inaccuracies incurred with physical ranging devices.

We consider two scenarios in both of which we vary the total number of nodes. In the first case (left graph in Figure 8), we use topologies in which all nodes are localizable given a sufficient number and placement of initial beacon nodes, and calculate the average localization error for a given number of nodes. The average localization error in Kairos is within the same order shown in [2, Figure 9], thereby confirming that Kairos is competitive here. Note that this error decreases with increasing network size as expected because the Gaussian noise introduced by ranging is decreased at each node by localizing with respect to multiple sets of ranging nodes and averaging the results. In the second scenario (right graph in Figure 8), we vary the percentage of initial beacon nodes for the full 24 node topology, and calculate how many nodes ultimately become localizable. This graph roughly follows the pattern exhibited in [2, Figure 12], thereby validating our results again.
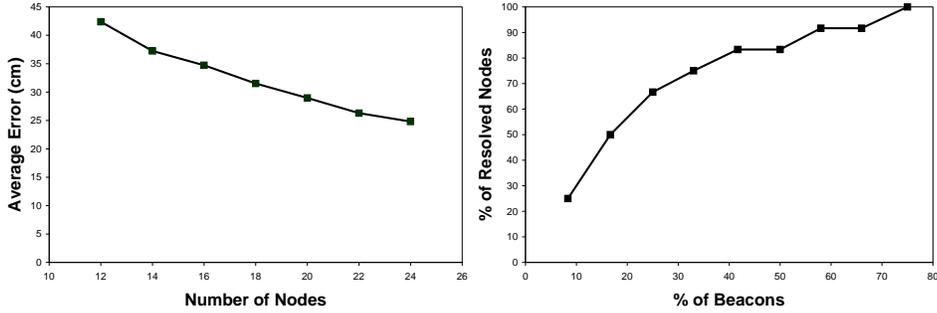
Fig. 8. Average Error in Localization (L) and Localization Success Rate (R)

Vehicle Tracking: For this purpose, we use the same vehicle tracking parameters as used in [14] (for grid size, vehicle speed, sound RMS, acoustic sensor measurement simulations, sensor placement and connectivity, and Direction-of-Arrival sensor measurements) for comparing how Kairos performs against [14]. A direct head-to-head comparison against the original algorithm is not possible because we have fewer nodes than they have in their simulations, so we present the results of the implementation in a tabular form similar to theirs. We simulate the movement of a vehicle along the Y-axis at a constant velocity. The motion therefore perpendicularly bisects the X-axis.

We do two sets of experiments. The first one is to measure the tracking accuracy as denoted by the location error ($\|\hat{x}_{MMSE} - x\|$) and its standard deviation ($\|\hat{x} - \hat{x}_{MMSE}\|^2$) as well as the tracking overhead as denoted by the belief state as we vary the number of sensors (K). The main goal here is to see whether we observe good performance improvement as we double the number of sensors from 12 to 24. As table 1 shows, this is indeed the case: the error, error deviation, and exchanged belief state all decrease, and in the expected relative order.

K	Avg $\|\hat{x}_{MMSE} - x\|$	Avg $\|\hat{x} - \hat{x}_{MMSE}\|^2$	Avg Overhead (bytes)
12	42.39	1875.47	135
14	37.24	1297.39	104
16	34.73	1026.43	89
18	31.52	876.54	76
20	28.96	721.68	67
22	26.29	564.32	60
24	24.81	497.58	54

Table 1. Performance of Vehicle Tracking in Kairos

In the second experiment, we vary the percentage of sensor nodes that are equipped with sensors that can do Direction-of-arrival (DOA) based ranging, and not just direction-agnostic circular acoustic amplitude sensors. These results are described in [13].

5   Conclusion and Future Work

This paper should be viewed as an initial exploration into a particular model of macro-programming sensor networks. Our contribution in this paper is introducing, describing, and evaluating this model on its expressivity, flexibility, and real-world performance metrics. Kairos is not perfect in that, at least in its current incarnation, it does not fully shield programmers from having to understand the performance and robustness implications of structuring programs in a particular way; nor does it currently provide handles to let an application control the underlying runtime resources for predictability, resource management, or performance reasons. Finally, while Kairos includes a middleware communication layer in the runtime service that shuttles serialized program variables and objects across realistic multihop radio links, today, this layer lacks the ability to optimize communication patterns for a given sensornet topology. Therefore, we believe that Kairos opens up several avenues of research that will enable us to explore the continuum of tradeoffs between transparency, ease of programming, performance, and desirable systems features arising in macroprogramming a sensor network.

References

1. J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *OSDI, 2002*.
2. A. Savvides, C. Han, and S. Srivastava, "Dynamic Fine-Grained localization in Ad-Hoc networks of sensors," *MOBICOM, 2001*.
3. A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," *SenSys, 2003*.
4. N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," *SenSys, 2004*.
5. A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring," *WSNA, 2002*.
6. A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao, "Habitat monitoring: application driver for wireless communications technology," *SIGCOMM Comput. Commun. Rev., 2001*.
7. K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," *MobiSys, 2004*.
8. M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," *NSDI, 2004*.
9. S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong, "TAG: A tiny AGgregation service for ad-hoc sensor networks," *OSDI, 2002*.
10. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The design of an acquisitional query processor for sensor networks," *SIGMOD, 2003*.
11. W. F. Fung, D. Sun, and J. Gehrke, "Cougar: the network is the database," *SIGMOD, 2002*.
12. S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.
13. R. Gummadi, O Gnawali, and R Govindan,   "Macro-programming Wirless Sensor Networks using Kairos,"   *USC CS Technical Report Number 05-848*, 2005, http://www.cs.usc.edu/Research/ReportsList.htm.
14. J. Reich J. Liu and F. Zhao, "Collaborative in-network processing for target tracking," *EURASIP, 2002*.

15. J. Liu, M. Chu, J. Liu, J. Reich, and F.Zhao, "State-centric programming for sensor and actuator network systems," *IEEE Perv. Computing, 2003*.

16. E. Cheong, J. Liebman, J. Liu, and F. Zhao, "Tinygals: a programming model for event-driven embedded systems," *SAC, 2003*.

17. T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood, "Envirotrack: Towards an environmental computing paradigm for distributed sensor networks," *ICDCS, 2004*.

18. A. Bakshi, J. Ou, and V. K. Prasanna, "Towards automatic synthesis of a class of application-specific sensor networks," *CASES, 2002*.

19. A. Bakshi and V. K. Prasanna, "Algorithm design and synthesis for wireless sensor networks," *ICPP, 2004*.

20. R. Kumar, M. Wolenetz, B. Agarwalla, J. Shin, P. Hutto, A. Paul, and U. Ramachandran, "Dfuse: a framework for distributed data fusion," *SenSys, 2003*.

21. R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," *DMSN, 2004*.

22. http://www.cs.berkeley.edu/projects/parallel/castle/split-c/

23. A. Boulis, C. Han, and M. B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," *MobiSys, 2003*.

24. B. Greenstein, E. Kohler, and D. Estrin, "A sensor network application construction kit (SNACK)," *SenSys, 2004*.

25. J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," *SenSys, 2004*.

26. P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *ASPLOS-X, 2002*.

27. T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi, "Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet," *MobiSys, 2004*.

28. D. B. Terry, M. M. Theimer, K. Petersen, Demers Demers, M. J. Spreitzer, and C. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," pp. 172–183, Dec. 1995.

29. P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," *NSDI, 2004*.

30. Douglas S. J. De Couto, Daniel Aguayo, John Bicket, and Robert Morris, "A high-throughput path metric for multi-hop wireless routing," in *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom '03)*, San Diego, California, September 2003.

31. Guido van Rossum and Fred L. Drake Jr., editors, "Extending and embedding the python interpreter," http://docs.python.org/ext/ext.html.

32. http://www.xbow.com/Products/XScale.htm.

33. J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin, "Emstar: An environment for developing wireless embedded systems software," *CENS-TR-9, 2003*.

34. http://www.xbow.com/Products/productsdetails.aspx?sid=72.

35. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGOPS Oper. Syst. Rev., 2000*.

36. W. Ye and J. Heidemann, "Medium access control in wireless sensor networks," *ISI-TR-580, 2003*.

37. http://www.xbow.com/Products/productsdetails.aspx?sid=73.

38. J. Heidemann, F. Silva, and D. Estrin, "Matching data dissemination algorithms to application requirements," *SenSys, 2003*.