Kensho: A Dynamic Tasking Architecture for Sensor Networks

James Horey
Department of Computer Science
University of New Mexico
jhorey@cs.unm.edu

Arthur B. Maccabe
Department of Computer Science
University of New Mexico
maccabe@cs.unm.edu

Angela Mielke
Los Alamos National Laboratory
amielke@lanl.gov

Abstract

Recent research on sensor network programming architectures has managed to alleviate many common programming burdens, but has not fully addressed the problems associated with tasking and deployment. This has impeded the development of new multi-purpose sensor networks that require new software abstractions and mechanisms.

We present the design of a new software architecture for sensor networks that provides abstractions to explicitly address the issues of tasking and deployment. We also present an initial implementation of our architecture in simulation to investigate the potential memory overheads. Finally, we present a mobile tracking application to demonstrate how to use the abstractions provided by our architecture.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design Languages

1 Introduction

New application scenarios, such as large-scale urban sensing, are driving sensor networks in new directions. One direction is the move from dedicated, single-purpose networks to open, multi-purpose sensor networks[22]. Unlike single-purpose sensor networks[17][14][8], these multi-purpose networks will amortize deployment costs by supporting multiple applications provided by many users. This additional functionality requires more flexible tasking abstractions than provided in existing architectures.

In general, multi-purpose sensor networks impose the following requirements:

- Tasking - The process of *mapping* a set of functions onto a set of nodes. The user must be able to define the set of sensor nodes flexibly and easily.
- Dynamic Retasking - The process of *remapping* the set of functions running on a set of nodes following some internal or external event.
- Redeployment - Mechanisms to handle the removal and introduction of sensor nodes gracefully are necessary.

Existing sensor network architectures do not fully address these tasking requirements and, as such, are inadequate with respect to next generation, multi-purpose sensor networks. New tools and architectures that recognize and solve these deficiencies are necessary.

We present the design of a communications and tasking architecture called *Kensho*. Kensho provides abstractions and mechanisms to explicitly address the issues associated with tasking multi-purpose sensor networks. The Kensho architecture extends common abstractions such as group specification, task assignment, and name-based communication, and uniquely combines these abstractions to address the previously outlined requirements.

2 Overview and Abstractions

The Kensho architecture relies on the following key abstractions:

- Explicit group specification
- Hierarchical task assignment
- Hierarchical tuple-based communication

Kensho's explicit group specification allows the user to spatially and temporally divide the sensor network. Hierarchical task assignment then uses the explicit group specifications to map functions onto the sensor network. Functions are tasked either to run on individual group members or to run collectively on the group. Finally, this tasking structure is used to provide tuple-based communication with a hierarchical namespace. Data is sent and received according to the role assigned to the function.

Limited versions of these abstractions have been explored in the literature[21][18][3]. However, previous works have used these abstractions in isolation, limiting the applicability of these systems in addressing all the tasking requirements. The Kensho architecture expands upon these initial ideas and uniquely combines these abstractions to create a more flexible tasking architecture.

2.1 Explicit Group Specification

Explicit group specification is used to functionally divide the sensor network into different roles. Such roles include simple tasks such as data collection and aggregation, and complex tasks such as object classification and tracking. Currently, many systems[7][19] implicitly define roles; sensor nodes are defined by the code they currently run. Explicit functional roles allow us to use groups in conjunction with other abstractions such as hierarchical task assignment and communication.

Groups are defined in the Kensho architecture using the following user provided information:

- Admission Function
- Potential Candidate List

An admission function is a binary function, unique to each group, that is executed by sensor nodes to determine group membership. The admission function is capable of accessing local storage and sensor data using the API described in Section 3. The ability to access the local storage and sensor devices allows membership in a group to be defined by a wide array of data conditions. For example, a simple admission function may read one or more sensor values, perform some simple computation on the data, and return *true* if the result exceeds some threshold.

The user is able to limit which sensor nodes run the admission function by specifying a candidate list. The candidate list contains the set of unique IDs of the sensor nodes that should execute the specified admission function.

The Kensho architecture allows the user to specify two group types: *static* and *dynamic*. These group types correspond respectively to spatial and temporal divisions in labor, and differ in when the admission function is executed. For static groups, the admission function is run once when the sensor node receives the admission function. A *true* return value indicates that the sensor node should join the group. Otherwise, the sensor node should not join the group. Membership in static groups does not change over time. This allows static groups to be used for spatial division of labor. For example, sensor nodes equipped with passive infrared devices could be statically grouped and would run analysis functions that differ from functions running on nodes equipped only with thermistor devices.

For dynamic groups, the admission function is run periodically throughout the lifetime of the sensor node. As such, one of four situations can occur:

- The admission function returns *true* and the sensor node currently is not a member of the group. The sensor node then joins the group.
- The admission function returns *true* and the node is already a member. The sensor node then continues being a member.
- The admission function returns *false* and the node is already a member. The sensor node then leaves the group.
- The admission function returns *false* and the node is not a member. The sensor node does nothing.

Because dynamic group membership can change over time, dynamic groups allow the sensor network to be tasked temporally. This can be used, for example, to activate analysis functions that only run when an object is near a group of sensors. These functions would automatically stop running once the object moves away from the sensors.

2.2 Hierarchical Task Assignment

Task assignment is the process by which a set of functions is mapped to a set of sensor nodes. By leveraging explicit group specification, functions are mapped to functional groups instead of specific sensor nodes. This makes the system more robust to individual sensor node changes; as new nodes are introduced into the network and old nodes are removed, the Kensho architecture can remap the tasked functions onto alternative sensor nodes associated with the group. In the case of dynamic groups, tasked functions stop executing when the sensor node leaves the group. Unlike other software architectures[9], tasked functions in Kensho consist of normal *C* functions that are not restricted to predefined tasking instructions. Examples of functions and automatic function remapping is given in Section 4.

The Kensho architecture provides two methods to task a group: *collective* and *local*. Locally tasked functions are run on the group members while collectively tasked functions are run on the group. These tasking abstractions are similar to the process of abstracting for-loops with a functional *map* operation, where instead of applying the same function iteratively to each data value, the function is applied to the entire list. Code that samples and reports sensor data can be locally tasked to the appropriate group without specifying individual sensor nodes. Similarly, instead of assigning code that aggregates data to run on a pre-defined leader node, the user can specify that this code be tasked as a collective function. Because the user does not choose which sensor node performs the collective operation, the Kensho runtime has the flexibility to choose from multiple group leader policies. For instance, collective functions could be implemented to run on a single powerful node, or rotate between multiple sensor nodes. Although the current algorithm explored in Section 3 uses a single leader election scheme, exploring alternative group leader policies remains a subject for future work.

2.3 Hierarchical Communication

By taking advantage of its hierarchical tasking scheme, Kensho is able to abstract common communication patterns by providing the following communication mechanisms:

- All communication consists of accessing named data tuples
- Data tuples are accessed strictly according to the task structure
- Sensor devices on nodes are accessed using the same communication methods

These abstractions closely match the data-centric nature of sensor network applications and alleviates the need for the programmer to manage network messages. In order to provide these abstractions, each group, along with individual sensor nodes, contains its own storage structure. Locally tasked functions access the storage associated with a particular group member, while collectively tasked functions access a separate group storage. Data tuples can only be placed into the group storage by locally tasked functions using a *publish* command. Likewise, collectively tasked functions can only place data items into the local storage of group members by performing a *push* command. These operations abstract many-to-one and one-to-many communication patterns and are similar to the patterns explored by Chu *et al.*[4].

The Kensho architecture also allows interaction with the underlying sensor devices using the same communication methods. Each sensor node maintains a list of reserved keywords that specify particular sensor devices. For instance, the string *magnetometer* interfaces with the magnetometer

Function	*RAM Overhead (bytes)*
Group Management	1284
Thread Management	956
Radio Management	84
Total	2324

Figure 1. The table shows the RAM usage for the major Kensho components.

device. Upon accessing a data tuple with that name, the Kensho architecture redirects the request to the appropriate sensor device. The sensor device then returns a data item containing the sensor reading.

By employing a single abstraction for both sensor data and stored data, a single function specification is able to serve multiple roles. A function referring to *thermistor* accesses the thermistor device if locally tasked on a node. If the same function is tasked as a collective function on a group of nodes, the data is fetched from the group store.

3 Preliminary Implementation

We have implemented the Kensho architecture using an in-house sensor network system simulator. The purpose of this implementation is to demonstrate the usefulness of our architecture and to investigate preliminary memory overheads. Measuring detailed communication overhead is a subject of future work. As such, our simulator provides a system-level view of the sensor nodes. Each node is simulated by a process linked with the Kensho library. Wireless communication is handled by a separate broadcast daemon. An external environmental simulator is used to provide sensory data. Although we could have used other available simulators, such as NS-2[1] or TOSSIM[15], the purpose of the initial implementation justified using our simulator for simplicity.

All user functions, including the admission functions, are written in *C*. The Kensho abstractions are implemented as a set of two *C* libraries, one for grouping and tasking and another for communication, that can be used by user functions. Sensor nodes that are responsible for tasking, typically connected to the user's laptop, employ the grouping and tasking library. Otherwise, sensor nodes that are being tasked are only linked with the communication library. We assume each sensor node runs an operating system linked with one of these Kensho libraries.

All code was compiled using *gcc 3.4* with the *Os* compiler flag for the *Pentium* architecture. The current implementation consumes a total of 2.27 *kb* of data memory (RAM - BSS and DATA sections). Our implementation also consumes 14.2 *kb* of static program memory (ROM - TEXT section) for individual nodes. Figure 1 breaks down the memory overhead by major Kensho components. Current platforms such as the TelosB contains 48 *kb* of program flash memory and 10 *kb* of data memory and can accommodate the current memory overheads. Further code optimization and removal of redundant functions and datastructures should reduce memory overheads.

[1] www.isi.edunsnamns

3.1 Group and Task Implementation

The grouping and tasking library includes methods to create both static and dynamic groups. The grouping methods, *new_static_group* and *new_dynamic_group*, accept a pointer to the admission function, along with the an array representing the candidate list. This array contains the unique IDs of the candidate sensor nodes. The library also includes a method, *compute*, that provides the tasking capabilities. The *compute* method accepts a group ID identifying the group, an identifier for the function that is to be run on the group, and a flag indicating whether the function should be tasked collectively or locally. This API is summarized in the top figure of Figure 2.

In order to use these methods, the user must construct a simple *C* program linked with the grouping and tasking library. Once compiled, all grouping and tasking instructions are translated into a series of network messages. These messages, along with the object files containing the tasked functions, are then propagated onto the simulated sensor network.

In our current implementation, each group elects a single leader to execute all collective functions. When a sensor node joins a group, it initiates a leader election protocol. In our election protocol, each node maintains a record of the current leader. Initially, each node records its own ID as the current leader. It then floods an election message that contains its own ID. Upon receiving a leader election message, the sensor node inspects the message and if the message has a larger ID value than the current leader, the node updates its leader value and then retransmits the message with the larger ID. This continues until all the nodes in the group converge upon a common answer. Currently, each node assumes that a common answer has been achieved if it has not received a new election message in some short period of time. Upon converging on a common leader, a message is sent to that node electing it as the leader. Afterwards, the leader floods a keep-alive beacon to all the members of the group.

When a new leader is selected (old leaders may leave the group or new sensor nodes may be introduced), the previous leader is able to initiate a leader-handoff protocol with the new group leader. This protocol transfers data contained in the group storage to the new leader, but does not transfer the current state of the collective function. In our current implementation, all group members cache the collective functions associated with their group.

3.2 Storage and Sensor Access

The hierarchical communication and storage library contains methods to access both the local and group storage, along with the sensor devices. Unlike the grouping and tasking instructions, which are concerned with the entire sensor network, these instructions are used by the tasked functions to exchange data within defined groups. The API is summarized in the bottom table of Figure 2.

The local and group storage is currently implemented as a hash table. Data items are accessed via simple string-based names and stored as arrays. These methods are synchronous and return a list of data items. If the user attempts to access

Tasking API	Description
new_static_group	Takes in a list of IDs and forms a new static group.
new_dynamic_group	Takes in an admission function and forms a new dynamic group.
compute	Associates groups with a set of functions.

Local API	Description
publish_data	Stores named data in the group collective store.
collect_data	Retrieves named data either from the local store or the sensor devices.
remove_data	Removes the named data from the local store.

Figure 2. The collective API describes the methods available for group creation and tasking. The local API describes the methods available for communication and data access.

non-existent data, the method returns an empty list.

The data access method, *collect_data*, also accesses the local sensor devices for locally tasked functions running on sensor nodes. Each sensor device registers a name during device initialization. Accessing data with a sensor device name activates and reads data from the appropriate sensor device. Collectively tasked functions accessing data with the name of a sensor device instead accesses the group storage.

3.3 System Requirements

Kensho is not designed as a stand-alone system and requires basic functionality from an underlying operating system. We assume that the operating system is capable of loading and unloading new application modules. Preliminary work in dynamically linked application modules[11][5] demonstrate this as a reasonable assumption. Due to the abstractions that Kensho provides, operating systems that export a thread-like interface for application functions are more suitable[6][2]. The Kensho architecture also assumes an underlying messaging protocol. Although our simulation currently employs a flooding protocol, we are investigating more efficient protocols for data routing[13][12] and code propagation[16].

4 Mobile Object Tracking

We developed a mobile-object simulator and an associated tracking application using the Kensho architecture. This application benefits from the Kensho architecture by using *dynamic groups* to task functions that continue to execute even as the group membership changes. The admission function reads the magnetometer value and returns *true* when the value is positive, otherwise it returns *false*. The admission function superficially resembles the left-hand function in Figure 3. However, instead of publishing the collected data, the admission function returns *true*. This admission function ensures that as the object moves away from a sensor node, the node leaves the group and stops processing.

In our simulation, an object starts at position (5, 5) and moves in a clockwise, circular fashion at a velocity of 1.35 units per time interval. Sensor nodes are placed in 1 unit increments in a grid layout. When the object is within a predefined radius of a magnetometer sensor, the sensor registers a positive value. Sensor nodes within the sensor radius employ a centroid algorithm similar to the one presented by Welsh *et al.*[20] to estimate the position of the object. Since our architecture does not focus on the actual programming methods, the details of the centroid algorithm are omitted.

The locally tasked function reads the magnetometer sensor (line 1 of Figure 3) and publishes extreme data values to the group storage (line 3). All sensor nodes within the radius of detection runs the locally tasked function. The collective function, in turn, collects the data published by the nodes (line 1) and performs the centroid algorithm using that data (line 2). Finally, the collective function publishes the result to the user (line 3).

Figure 4 shows that as new sensor nodes join the dynamic group, they automatically begin executing the locally tasked functions. The collectively tasked function also automatically executes on the group leader even as group leaders change. Although the application was run in simulation, initial results indicate that dynamic groups can be used to accurately track mobile objects.

5 Related Work

Kensho builds upon previous work in programming and communication architectures. The Tenet architecture[9] separates mote-level functionality from basestation functionality by restricted the tasking language used on the sensor nodes. Similarly, the Kensho architecture abstracts the notion of code running on a basestation from code running on individual nodes. However, sensor nodes are not restricted by a specialized tasking language and users are encouraged to employ the idea of *collective* and *local* functions on groups. In this light, the Tenet architecture can be viewed as a specific implementation of some of the Kensho abstractions.

EnviroTrack[1] is a programming tool that uses mechanisms similar to Kensho's dynamic group. EnviroTrack, however, is primarily designed for mobile object tracking applications while Kensho addresses a wider array of applications. Kairos[10] is a macroprogramming tool that shares similarities with Kensho's collective tasking abstractions. However, Kairos does not explicitly address tasking and requires user programs to address specific nodes.

SINA[19] employs automatic clustering of sensor nodes based on power-level and proximity. Groups of sensor nodes that rely on other attributes, such as shared sensor readings, can only be defined implicitly within a query.

Agent-based systems, such as Agilla[7] also implicitly defines groups by the nodes that the agents decide to visit. This makes it difficult to change the set of nodes the agent should execute on without modifying the code contained

```
    for(;;) {
[1]         data = collect_data("magnetometer");
[2]         if(data.value > THRESHOLD)
[3]             publish_data("location", data.value);
[4]         sleep(NODE_INTERVAL);
    }
```

```
for(;;) {
[1]     data = collect_data("location");
[2]     centroid = polygon_average(data.value, data.num_data);
[3]     publish_data("centroid", centroid);
[4]     remove_data("location");
[5]     sleep(COLLECT_INTERVAL);
```

Figure 3. The locally tasked function on the left collects environmental data using a *collect_data* call to the Kensho Node API. It then publishes that data to be consumed by the collective computation. The collective computation, shown on the right, collects the data from the nodes using a *collect_data* and calculates the centroid.

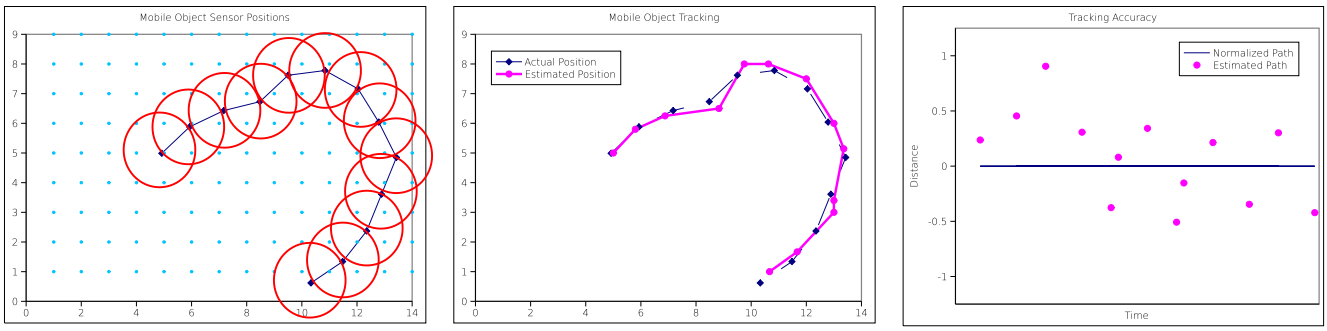

Figure 4. The circles indicate which sensors are in the dynamic group as the object moves in a clockwise direction. The horizontal line in the right figure represents the normalized path of the object, while the points indicate how far from the actual object the algorithm predicted the object to be.

within the agent itself.

Hoods[21] and Abstract Regions[20] are both neighborhood based programming abstractions and provide complementary services to Kensho. While Kensho uses groups to task the sensor network, Hoods and Abstract Regions are concerned with the communication and topological structure of these groups for programming purposes.

Römer *et al.* have explored frameworks for role assignment in sensor networks[18]. Our work operates at a lower level in the software stack and provides complementary services. Finally, Melete[22] is a recent system that supports concurrent applications. Their work, however, operates at the node level and is not concerned with higher level tasking abstractions.

6 Conclusion and Future Work

In this paper, we presented the design and initial implementation of a communications and tasking architecture for sensor networks. By extending and combining programming and communication abstractions explored in existing systems, our architecture is able to task and deploy multipurpose sensor networks gracefully. Specifically, our architecture tasks the sensor network both spatially and temporally using explicit group specification. Functions can be tasked to run collectively or locally on these groups; this abstracts the division in labor found in other systems. Our architecture takes advantage of a hierarchical tasking scheme to abstract data storage and communication between collective and local functions. Results from a mobile object tracking application demonstrate that the abstractions provided by our system are useful and can simplify application tasking. These initial results indicate that our architecture aids in the construction and deployment of realistic sensor network applications.

We are in the process of porting the Kensho architecture to run on existing sensor network platforms, such as the TMote Sky[2]. The memory overheads from our prototype implementation indicate that a port to such resource constrained platforms is viable. We expect to test the communication overheads associated with tasking and data storage access once this port is complete.

We are also exploring alternate group leader policies and distributed group implementations to improve system robustness. Because the user does not specify the sensor node collective functions are executed on, we expect to test these alternate leader policies without affecting the correctness of user programs.

6.1 Acknowledgments

This paper is supported by the U.S. Department of Energy/NNSA and Los Alamos National Laboratory under contract DE-AC52-06NA25396 and approved for public release under LA-UR-06-6041.

7 References

[1] 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. In *ICDCS*, 2004.

[2] www.moteiv.com

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, and R. Han. Mantis: System support for multimodal networks of in-situ sensors. In *WSNA*, 2003.

[3] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[4] D. Chu, K. Lin, A. Linares, G. Nguyen, and J. Hellerstein. Sdlib: A sensor network data and communications library for rapid and robust application development. In *IPSN/SPOTS*, 2006.

[5] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Runtime dynamic linking for reprogramming wireless sensor networks. In *SenSys*, 2006.

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *SenSys*, Nov. 2006.

[7] C.-L. Fok, G.-C. Roman, and C. Lu. Mobile agent middleware for sensor networks: An application case study. In *IPSN*, 2005.

[8] L. Girod, M. Lukac, V. Trifa, and D. Estrin. The design and implementation of a self-calibrating distributed acoustic sensing platform. In *SenSys*, 2006.

[9] R. Govindan, E. Kohler, D. Estrin, F. Bian, K. Chintalapudi, O. Gnawali, R. Gummadi, S. Rangwala, and T. Stathopoulous. Tenet: An architecture for tiered embedded networks. Technical report, Center for Embedded Networked Sensing, 2005.

[10] R. Gummadi, O. Gnawali, and R. Govindan. Macroprogramming wireless sensor networks using kairos. In *DCOSS*, 2005.

[11] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor nodes. In *MobiSys*, 2005.

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

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *MobiCom*, 2000.

[14] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.-S. Peh, and D. Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experience with ZebraNet. In *ASPLOS*, 2002.

[15] P. Levis. Tossim: Accurate and scalable simulation of entire tinyos applications. In *SenSys*, 2003.

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

[17] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *WSNA*, 2002.

[18] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic role assignment for wireless sensor networks. In *11th ACM SIGOPS European Workshop*, 2004.

[19] P.-C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor Information Networking Architecture and Applications. *IEEE Personel Communication Magazine*, August 2001.

[20] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[21] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys*, 2004.

[22] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys*, 2006.