

CHAPTER 1

INTRODUCTION

Networked sensor systems

A networked sensor system (a “sensor network”) is a distributed computing system where some or all nodes are capable of interacting with the physical environment. These nodes are termed as sensor nodes and the interaction with the environment is through sensing interfaces. Sensors typically measure properties such as temperature, pressure, humidity, flow, etc., when sampled. The sensed value can be one-dimensional or multi-dimensional. Sensor networks have a wide range of applications. Acoustic sensing can be used to detect and track targets in the area of deployment. Temperature, light, humidity, and motion sensors can be used for effective energy management through climate moderation in homes and commercial buildings.

Wireless sensor networks (WSNs) [44, 3, 17] are a new class of sensor networks, enabled by advances in VLSI technology and comprised of sensor nodes with small form factors, a portable and limited energy supply, on-board sensing, computing, and storage capability, and wireless connectivity through

a bidirectional transceiver. WSNs promise to enable dense, long-lived embedded sensing of the environment. The unprecedented degree of information about the physical world provided by WSNs can be used for in situ sensing and actuation. WSNs can also provide a new level of context awareness to other back-end applications, making sensor networks an integral part of the vision of pervasive, ubiquitous computing—with the long-term objective of seamlessly integrating fine grained sensing infrastructure into larger, multi-tier systems.

There has been significant research activity over the last few years in the system-level aspects of wireless sensing. System level refers to the problems such as: (a) localization [41] and time synchronization [15, 16] to provide the basic “situatedness” for a sensor node; (b) energy-efficient medium access protocols that aim to increase the system lifetime through means such as coordinated sleep-wake scheduling [60]; (c) novel routing paradigms such as geographic [33, 47], data-centric [22], and trajectory-based [40] that provide the basic communication infrastructure in a network where the assignment and use of globally unique identifiers (such as the IP addresses of the Internet) is infeasible or undesirable; (d) modular, component-based operating systems for extremely resource constrained nodes [27], etc. A variety of routing and data fusion protocols for generic patterns such as multiple-source single-sink data gathering trees are also being developed to optimize for a range of goodness metrics [30, 29, 61]. A comprehensive overview of state of the art in system level aspects of wireless embedded sensing can be found in [31, 18].

1.1 SENSOR NETWORKS AND TRADITIONAL DISTRIBUTED SYSTEMS

It is instructive to compare and contrast the fundamental nature of networked sensing with traditional parallel and distributed computing, with a view to identifying the degree to which the research in the latter field over the past few decades can be leveraged (with or without modification) to propose solutions for analogous problems in the former. Since the primary focus of this work is on models and methodologies for programming of large-scale networked sensor systems, the comparison will be biased towards aspects which influence application development and not so much on system level issues.

Sensor networks are essentially collections of autonomous computing elements (sensor nodes) that pass messages through a communication network and hence fit the definition of a distributed computing system proposed in [8]. However, some of the fundamental differences between networked sensor systems and traditional distributed computing systems are as follows:

Transformational versus reactive processing

The primary reasons for programming applications for a majority of traditional distributed computing systems were “high speed through parallelism, high reliability through replication of process and data, and functional specialization” [8]. Accordingly, the objective of most programming models and languages was to (i) allow the programmer to expose parallelism for the compiler and runtime system to exploit and (ii) provide support for abstractions such as shared memory that hide the distributed and concurrent nature of the underlying system from the application developer. In other words, the purpose of most abstractions was to allow the programmer to still visualize the target architecture as a von Neumann machine, which provided an intuitive and straightforward mental model of reasoning about sequential problem solving. Alternate approaches such as dataflow and functional programming were also proposed, motivated by a belief in the fundamental unsuitability of the von Neumann approach for parallel and distributed computing [5]. Regardless of the approach, most parallel and distributed applications were ultimately transformational systems that are characterized by a function that maps input data to output data. This function can be specified as a sequential, imperative program for a von Neumann architecture, and the purpose of parallelizing and distributing the execution over multiple nodes is mainly to reduce the total latency.

A networked sensor system is not a transformational system that maps a well-defined set of input data to an equally well-defined set of output data. Instead, like a majority of embedded software, it is a continuously executing and primarily reactive system that has to respond to external and internal stimuli [24]. An event of interest in the environment triggers computation and communication in the network. A quiescent environment ideally implies a quiescent network as far as application level processing is concerned.

Space awareness

An embedded sensor network can be considered to represent a discrete sampling of a continuous physical space. In fact, an abstract model of a distributed sensor network can be defined and analyzed purely in terms of measurements of the space being monitored [32], without any reference to the network architecture. In contrast to traditional distributed computing where all compute nodes were basically interchangeable and the physical location of a particular computing element is not directly relevant from a programming or optimization perspective, space awareness [63] is an integral part of embedded net-

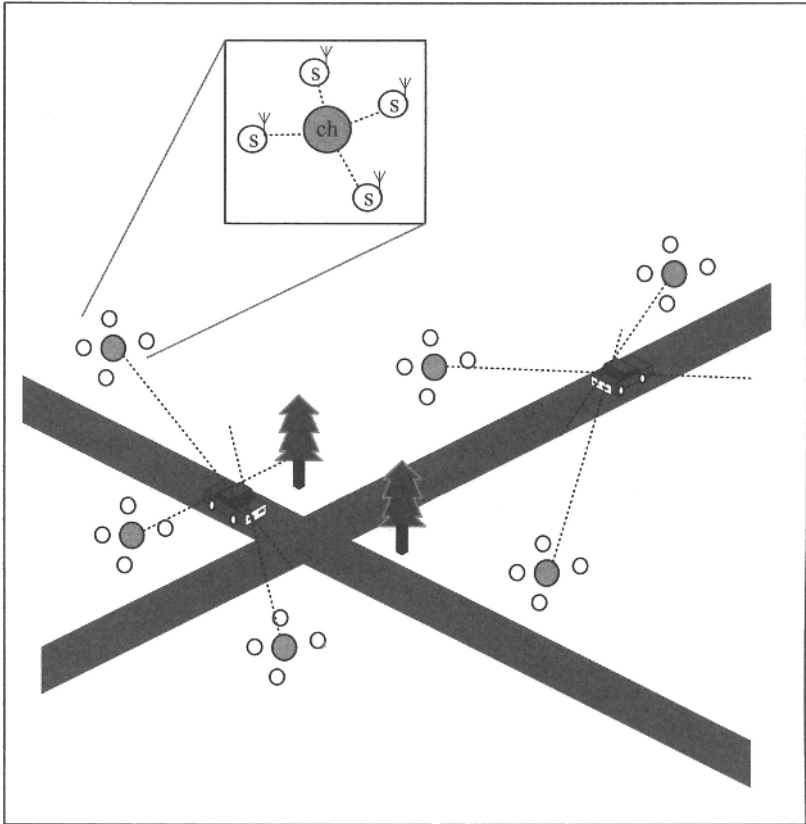


Figure 1.1 An example sensor network deployment for vehicle detection and tracking. Sensor nodes are deployed in clusters, with each cluster consisting of a relatively powerful clusterhead node and four resource-constrained sensor nodes. Each sensor could be equipped with acoustic and/or magnetic sensors. The individual sensor nodes in each cluster communicate their readings to the clusterhead which computes the line of bearing and possibly the type of vehicles. This information will be relayed to a supervisor station that can triangulate the object position by ending line of bearing estimates from multiple clusters. This particular scenario was one of the early use cases for wirelessly networked sensor systems.

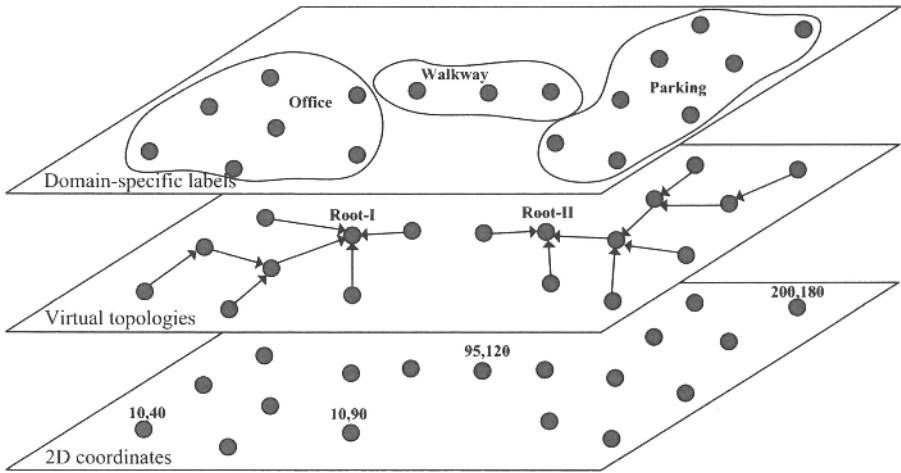


Figure 1.2 Multiple coordinate systems on the same deployment.

worked sensing. Most of the data in a sensor network deployment are created through the act of sampling the sensing interface(s), and the time and location of the sampling are in most cases a necessary part of the description of the sampled data. The spatio-temporal origin of a data item also affects the quality and quantity of processing performed on it.

Space awareness implies the existence of a coordinate system in which sensor nodes can be situated. In fact, a typical sensor network deployment is likely to have more than one coordinate system, each designed for a different purpose. For instance, the absolute or relative geographic coordinates might be required for tagging data samples at the node level, whereas the routing protocols could be using a different coordinate system that leads to reduced congestion and higher probability of timely data delivery in the network. Yet another coordinate system could be used for back-end processing which maps a particular (x, y) coordinate to, say, a building, a corridor, or a warehouse, depending on the application domain. Figure 1.2 depicts three coordinate systems overlaid on the same sensor network. From the perspective of application development for networked sensor systems, a real or virtual coordinate system can be deemed to be an essential service included in the system level infrastructure, the details of which need not concern the programmer.

Another aspect of space awareness is that the application behavior can be naturally specified in terms of spatial abstractions than in terms of nodes and edges of the network graph. For example, a temperature monitoring

application can be specified as “if more than 70% of nodes within a 2-meter radius of any node report a temperature higher than 90 degrees, activate an alarm at that node location.” The deployment of the network itself can be specified in terms of the desired degree of coverage. The exact placement of sensor nodes might not be of interest to the application developers as long as the set of sensing tasks mapped onto a subset of those nodes at any given time collaboratively ensures the desired coverage. Space-aware specification of the desired functionality is a unique aspect of networked sensor systems that has no analogous equivalent in traditional parallel and distributed computing.

Nature of input data

A majority of the data in a networked sensor system represents the occurrence of events in the physical environment and/or carries information about the events. Each data instance can be considered as a first-class entity with associated properties that could change with time and distance from its point of origin. For instance, in embedded sense-and-respond systems where sensing is coupled with local actuation and timely response to detected events is essential, the utility of the data that represent occurrence of the event reduces with time. If the data are not processed by the application within a certain duration from its time of origin, it is effectively useless. In-network processing that seeks to move the computation close to the source of the data is required in many sensor network applications to guarantee the desired end-to-end functionality. This is in contrast to traditional distributed computing, where the distribution of data and placement of tasks on compute nodes is primarily determined by performance and reliability considerations.

Also, different subsets of the total data in the network will be of interest to different applications at a given time, or to the same application at different times. In a sensor network deployed for climate moderation in a commercial building, an application component that periodically logs all temperature readings in a central database might not be interested in the semantics of that information, whereas another application component that is responsible for maintaining a uniform climate could be interested in temperature gradients that are above a certain threshold. From a programming perspective, it is important to give application developers the freedom to define what is relevant and what is irrelevant and to produce and consume data at the desired level of semantic abstraction.

The semantics of data could also influence the protocols and services used for transporting data through the network, and for prioritizing in-network activities that are triggered in response to certain events. A piece of data that represents a catastrophic event such as a forest fire is much more important than any other data in the network at that time and the computation and communication resources in the network can be expected to be devoted to expediting the transmission of the forest fire notification to its eventual destination. In a purely transformational system, however, it can be argued that the notion of importance of a particular piece of data does not really exist.

1.2 PROGRAMMING OF DISTRIBUTED SENSOR NETWORKS

1.2.1 Layers of programming abstraction

Figure 1.3 depicts our view of the emerging layers of programming abstraction for networked sensor systems. Many protocols have been implemented to provide the basic mechanisms for efficient infrastructure establishment and communication in ad hoc deployments. These include energy-efficient medium access, positioning, time synchronization, and a variety of routing protocols such as data-centric and geographic routing that are unique to spatial computing in embedded networked sensing. Ongoing research, such as MiLAN [26], is focusing on sensor data composition as part of the basic infrastructure. Sensor data composition essentially means that the responsibility of interfacing with physical sensors and aggregating the data into meaningful application-level variables is delegated to an underlying runtime instead of being incorporated as part of the application-level logic. We now discuss the layers of abstraction from the highest level of abstraction to the lowest.

1.2.1.1 Service-oriented specification To handle the complexity of programming heterogeneous, large-scale, and possibly dynamic sensor network deployments and to make the computing substrate accessible to the non-expert, the highest level of programming abstraction for a sensor network is likely to be a purely declarative language. The Semantic Streams markup and query language [57] is an example of such a language that can be used by end users to query for semantic information without worrying about how the corresponding raw sensor data are gathered and aggregated. The basic idea is to abstract the collaborative computing applications in the network as a set of services and provide a query interpretation, planning, and resource management engine to translate the service requirements specified by the end

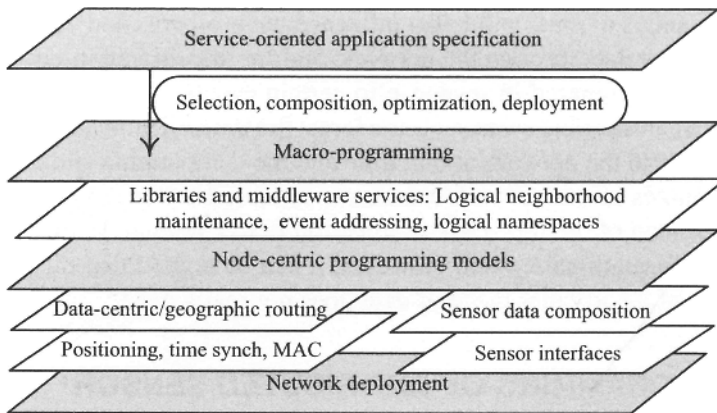


Figure 1.3 Layers of abstraction for application development on WSNs.

user into a customized distributed computing application that provides the result. A declarative, service-oriented specification allows dynamic tasking of the network by multiple users and is also easier to understand compared to low level distributed programming.

1.2.1.2 Macroprogramming The objective of macroprogramming is to allow the programmer to write a distributed sensing application without explicitly managing control, coordination, and state maintenance at the individual node level. Macroprogramming languages provide abstractions that can specify aggregate behaviors that are automatically synthesized into software for each node in the target deployment. The structure of the underlying runtime system will depend on the particular programming model. While service-oriented specification is likely to be invariably declarative, various program flow mechanisms—functional, dataflow, and imperative—are being explored as the basis for macroprogramming languages. Regiment [42] is a declarative functional language based on Haskell, with support for region-based aggregation, filtering, and function mapping. Kairos [23] is an imperative, control-driven macroprogramming language for sensor networks that allows the application developer to write a single centralized program that operates on a centralized memory model of the sensor network state. ATaG [6] (discussed in more detail in the remainder of this book) explores the dataflow paradigm as a basis for architecture-independent programming of sensor network applications.

```

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;
6:   //Initialization
7:   full_node_set=get_available_nodes();
8:   for (node temp=get_first(full_node_set); temp!=NULL;
9:        temp=get_next(full_node_set))
10:    self=get_local_node_id();
11:    if (temp==root)
12:      dist_from_root=0; parent=self;
13:    else dist_from_root=INF;
14:    neighboring_nodes=create_node_list(get_neighbors(temp));
15:    full_node_set=get_available_nodes();
16:    for (node iter1=get_first(full_node_set); iter1!=NULL;
17:         iter1=get_next(full_node_set))
18:      for(;;) //Event Loop
19:        sleep(sleep_interval);
20:        for (node iter2=get_first(neighboring_nodes); iter2!=NULL;
21:             iter2=get_next(neighboring_nodes))
22:          if (dist_from_root@iter2+1<dist_from_root)
23:            dist_from_root=dist_from_root@iter2+1;
24:            parent=iter2;

```

Figure 1.4 Kairos code example: Building a shortest path routing tree [23].

Figure 1.4 [23] is a complete, centralized Kairos program for building a shortest path routing tree from a root node that is an input parameter. The entire distributed algorithm for building such a tree is specified in this program. Note that this code is not directly executed on each node. Instead, it is parsed by a compiler that uses the program specification to (a) determine the actual code to generate for each of the nodes in the network and (b) manage the local and remote variables referred to in the code.

The initialization portion of the program gets all the nodes of the network, and for each node it sets the initial distance from root and the parent node pointer. The node that is to form the root of the routing tree sets its distance from root as zero and its parent pointer to itself, while all others set their distance to the root as infinity.

The event loop in lines 15 through 20 represents an iterative process where each node periodically contacts each of its one-hop neighboring nodes from the list of one-hop neighbors, determines if that node is closer to the root than itself, and conditionally sets its parent in the routing tree to the neighboring node that is nearest to the root.

```

let mesh = planarize world
nodesAbove =
  afilter ((>= threshold) .
    (read_sensor SENSTYP))
    mesh
midpoint nst1 nst2 =
  (read_nstate LOCATION nst1 +
    read_nstate LOCATION nst2) / 2
contourpoints node =
  let neighborsBelow =
    filter ((< threshold) .
      (read_nstate SENSTYP))
      (get_neighbors node)
  in map (midpoint (get_nstate node))
    neighborsBelow
all_contourpoints =
  amap contourpoints nodesAbove
in
  afold append all contourpoints

```

Figure 1.5 Regiment code example: Determining the contour between adjacent areas of a sensor network [42].

Figure 1.5 [42] provides a glimpse into the Regiment programming style. The program shown in the figure determines the contour between adjacent areas of the network, where the nodes on one side of the contour have sensor readings above some threshold. The program, written as a functional language, first prunes the network graph into a planar form (“planarize world”) and determines all the nodes whose sensor reading is above the threshold. The remainder of the code takes each node of the set of nodes above the threshold and forms a list of midpoints between the node and its neighboring nodes below the threshold. Finally, the list of midpoints generated at the contour nodes is aggregated to yield the contour line.

1.2.1.3 Node-centric programming In node-centric programming, the programmer has to translate the global application behavior in terms of local actions on each node, as well as individually program the sensor nodes using languages such as nesC [19], galsC [13], C/C++, or Java. The program accesses local sensing interfaces, maintains application level state in the local memory, sends messages to other nodes addressed by node ID or location, and responds to incoming messages from other nodes. While node-centric programming allows manual cross-layer optimizations and thereby leads to

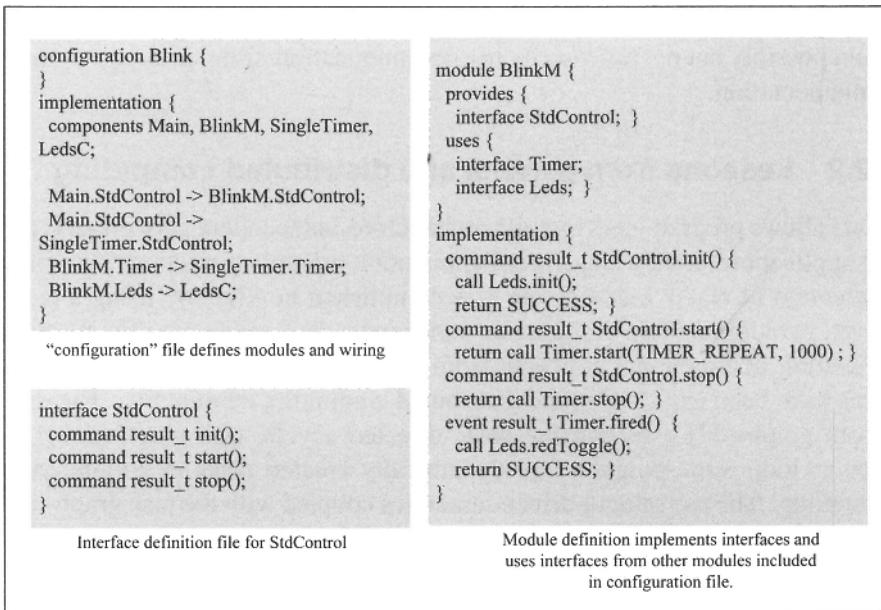


Figure 1.6 Programming in nesC.

efficient implementations, the required expertise and effort makes this approach insufficient for developing sophisticated application behaviors for large-scale sensor networks.

The concept of a **logical neighborhood**—defined in terms of distance, hops, or other attributes—is common in node-centric programming. Common operations upon the logical neighborhood include gathering data from all neighbors, disseminating data to all neighbors, applying a computational transform to specific values stored in the neighbors, etc. The usefulness and ubiquity of neighborhood creation and maintenance has motivated the design of node-level libraries [56, 55] that handle the low level details of control and coordination and provide a neighborhood API to the programmer.

Middleware services [26, 37, 62] also increase the level of programming abstraction by providing facilities such as phenomenon-centric abstractions. Middleware services could create virtual topologies such as meshes and trees in the network, allow the program to address other nodes in terms of logical, dynamic relationships such as leader–follower or parent–child, support state-centric programming models [35], etc. The middleware protocols themselves

will typically be implemented using node-centric programming models and could possibly but not necessarily use communication libraries as part of their implementation.

1.2.2 Lessons from parallel and distributed computing

ATaG allows programmers to write architecture-independent networked sensing applications using a small set of application-neutral abstractions. Intuitive expression of reactive processing is accomplished in ATaG by using a data-driven paradigm, while architecture-independence is made possible through separation of functional concerns from the nonfunctional. These two core ideas have been explored in the distributed computing community. The data driven graph [52] extended the basic directed acyclic task graph model to support loop representation and dynamically created tasks in parallel programming. The use of data-driven semantics coupled with the task graph-like representation enabled clarity and simplicity of program design, and it also allowed for some optimizations relating to the data communication between tasks.

The benefits of separating the core application functionality from other concerns such as task placement and coordination motivated the FarGo [28] model that enabled dynamic layout of distributed applications in large-scale networks where capabilities of nodes and links could vary at runtime. By explicitly indicating co-location and re-location semantics of the tasks, FarGo elevated the performance and reliability of applications by allowing the deferment of layout decisions to runtime. Distributed Oz [25] is perhaps the closest to ATaG in terms of its objective of network transparency and network awareness. Distributed Oz cleanly separates the application functionality from aspects of distribution structure, fault tolerance, resource control and security, and openness. There are no explicit operations to transfer data across the network. All invocations of `send()` and `receive()` are done implicitly through language constructs of centralized programming. IBM's PIMA project [9] explored a "write once, run anywhere" model for application front-ends by specifying device-specific presentation hints separately from the tasks and their interactions – yet again highlighting separation of functional and non-functional concerns as the key enabler of architecture independence.

Tuple space is an abstract computation environment that represents a global communication buffer accessible to computational entities in the system. This was the basis for the generative communication model in the Linda coordination language [20] and is also being applied in networked sensing [14]. Communication orthogonality is a property of generative communication

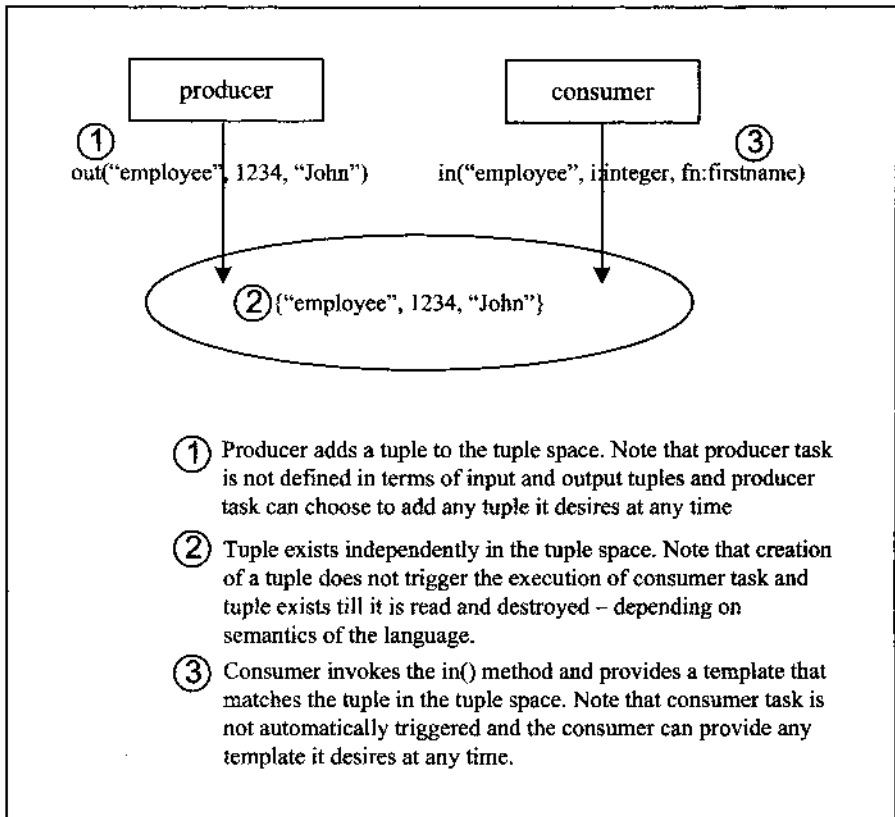


Figure 1.7 Programming with tuple spaces. The producer and consumer tasks communicate via “in” and “out” primitives. The tuple persists in the tuple spaces until it is actively retrieved by the consumer.

and means that both the sender and the receiver of a message are unaware of each other. ATaG also has this property because the tasks that produce and consume a particular data item in ATaG are not aware of each other. The data pool in ATaG is superficially similar to the notion of a tuple space. However, our active data pool moves the data items from producer to consumer(s) as soon as they are produced, and it schedules the consumer tasks based on their input interface and firing rules. This is different from the passive tuple space that merely buffers the produced data items and whose modifications are really a side effect of control-driven task execution.

In fact, the concept of tuple spaces has its roots in Blackboard architectures [43] of AI research. ATaG's active data pool is similar to the "demoned data servers" of DOSBART [34] that enabled distributed data-driven computation in a blackboard architecture. The notions of activity class and trigger activities of DOSBART are similar to the abstract tasks and their firing rules in the ATaG model, respectively.

1.3 MACROPROGRAMMING: WHAT AND WHY?

The primary focus of this dissertation is on the programming of large-scale networked sensor systems. The purpose of the typical sensor network deployment is to gather and process data from the environment for a single "end-to-end" objective. The program that executes on each node is part of a larger distributed application that delivers the results of an implicit or explicit domain specific query. Each node is required to be aware of its role in accomplishing the overall objective; that is, it is required to implement a predefined protocol for information exchange within the network. Consider a sensor network deployed for object tracking. The desired result of the implicit and perennial domain specific query in this case is the current location of target(s) (if any) in the network. A node-centric approach to programming the network requires each node to be programmed with the following behavior. The acoustic sensor is sampled periodically with a fixed or varying frequency, a Fourier transform is applied to the time-domain samples, and the result is compared with a set of acoustic patterns of interest to the end user. If a match is found, the time- and location-stamped result is communicated to a designated "clusterhead" node which performs further processing such as line of bearing estimation in an attempt to predict the location of the target.

This programming methodology where the desired global application behavior is manually decomposed by the programmer and subsequently coded into individual node-level programs is termed node-centric programming and is representative of state of the art. Node-centric programming has several limitations. Manual translation of global behavior into local actions is likely to be time-consuming and error prone for complex applications. If a new global behavior is to be added to an existing program, the modifications to the existing code are essentially ad hoc. The strong coupling of application-level logic and system-level services such as resource management, routing, localization, etc., also results in high coding complexity.

Macroprogramming broadly refers to programming methodologies for sensor networks that allow the direct specification of aggregate behaviors. The

existence of a mechanism to translate the macroprogram into the “equivalent” set of node-level behaviors is implicit. The exact interpretation of macroprogramming varies. A Regiment program specifies operations (such as *fold* and *map*) over sensor data produced by nodes with certain geographic or topological relationships of interest. Since these subsets of the global network state can be manipulated as a single unit, Regiment is a macroprogramming language. Kairos is a macroprogramming language because the programmer writes a single, centralized program for the entire network, and the compiler and runtime system are responsible for the translation of this program into node-level behaviors, and implementing data coherence, respectively. TinyDB also enables macroprogramming because the programmer who formulates the SQL-like declarative aggregate query over sensor data is not responsible for (or even aware of) the details of in-network processing that are responsible for data collection and processing.

We define the following two types of macroprogramming that are supported by ATaG.

- *Application-level macroprogramming* means that the programming abstractions should allow the manipulation of information at the *desired* level of semantic abstraction. The information may indicate the occurrence of an event and/or also carry information about the occurrence. For instance, in an object tracking application, the program should be able to access information such as “number of targets currently tracked,” “location of nearest target,” etc., without worrying about how that information is obtained.
- *Architecture-level macroprogramming* means that the programming abstractions should allow concise specification of common patterns of distributed computing and communication in the network. Such patterns are represented as part of neighborhood libraries defined for node-centric programming methodologies [55]. These will typically have equivalent, concise abstractions in the macroprogramming language whose node-level implementation invokes the libraries.

A macroprogramming language can be application-neutral or application-specific. The *application-specific* approach entails customized language features to support a particular class of networked sensing applications. For example, a programming language explicitly designed for multi-target tracking might provide the current set of target locations or the handles to the current targets as a language feature whose implementation is hidden from the user. A language for temperature monitoring might provide a topographic map of the

terrain as a built-in data structure that is created and maintained entirely by the runtime system. The advantage of this approach is that the implementation of domain-specific features can be optimized based on *a priori* knowledge of the pattern of information flow. If domain-specific features are integrated into the language, the resultant complexity of coding a behavior in that domain is also reduced. The drawback of this approach is that the portability and reusability of application-level code across network architectures, node architectures, and domains could be compromised. Also, adding new language features or modifying existing features might require a redesign of the runtime system and could be impossible or difficult for the application developer.

1.4 CONTRIBUTIONS AND OUTLINE

The two main contributions of this research are: (i) a **programming model** called the Abstract Task Graph (ATaG) for architecture-independent application development for a class of networked sensor systems and (ii) a component-based **software architecture for the runtime system**. A third contribution is a prototype environment for **visual programming** in ATaG and automatic **software synthesis** for the target network deployment. The prototype compiler integrated into this environment is designed to demonstrate functionally correct synthesis of a subset of the program features and does not optimize for any performance related metrics. Indeed, the definition of the compilation problem in the context of ATaG and the design and implementation of optimizing compilers for different scenarios is a significant research problem in its own right and one of the main areas of future work.

The Abstract Task Graph (ATaG)

ATaG is a macroprogramming model that builds upon the core concepts of data-driven computing and incorporates novel extensions for distributed sense-and-respond applications. In ATaG, the types of information processing functionalities in the system are modeled as a set of abstract tasks with well-defined input/output interfaces. User-provided code associated with each abstract task implements the actual processing in the system. An ATaG program is *abstract* because the exact number and placement of tasks and the control and coordination mechanisms are not defined in the program but are determined at compile-time and/or runtime, depending on the characteristics of the target deployment. Although ATaG is superficially based on the task graph representation, there are significant differences in the syntax and semantics, which

arise from the requirements of distributed networked sensing. The differentiating factors include the notion of “abstract” tasks and data items, the use of data-driven program flow semantics of the graph, the elevation of data items as a first class entity in the graph representation along with the computational tasks, the concept of spatial scope of a directed edge, etc.

There is a growing interest in defining macroprogramming languages [23, 42] and application development environments [12, 53] for sensor networks. ATaG enables a methodology for architecture-independent development of networked sensing applications. The same ATaG program may be automatically synthesized for different network deployments, or adapted as nodes fail or are added to the system. Furthermore, it allows application development to proceed prior to decisions being made about the final configuration of the nodes and the network, and in future implementations it will permit dynamic reconfiguration of the application as the underlying network changes.

ATaG provides *application-neutral* support for macroprogramming. Using a small set of basic abstractions, ATaG allows programmers to define their own semantics for tasks and data items. The modularity and composability of ATaG programs means that a library of common behaviors in a particular domain can be defined by the programmer and can later be plugged into other applications that need not know the implementation details of the library component. This approach provides the benefits of using predefined domain-specific features while avoiding the restrictiveness of a domain-specific, custom built runtime system.

Data-Driven ATaG Runtime (DART)

ATaG is supported by a runtime system called DART whose structure and function is not visible to the programmer. DART has a component-based software architecture for modularity and flexibility. Each component of DART provides one or more well-defined services to other components. The implementation of a service is hidden from the users of the service. The current DART design can be easily implemented on operating systems that support preemptive priority-based scheduling, multi-threaded execution, mutual exclusion semaphores, message queues, and other mechanisms to handle concurrent access to critical sections and coordinate interactions between threads. Most traditional operating system kernels provide these facilities. A prototype version of DART has been implemented in Java, and is designed to run on relatively heavy duty sensor nodes, although Java Virtual Machines for resource-constrained architectures are also available [48]. DART is also being implemented on the μ C/OS-II real-time OS kernel [39], which has been ported to a vast number of devices.

The performance of DART is unlikely to compare favorably with hand-optimized runtime systems where different functionalities are tightly integrated into an inflexible, monolithic structure, and many cross-layer optimizations are incorporated into the design. However, the tradeoff between usability and flexibility, on one hand, and hand-optimized performance, on the other, is common in all methodologies that seek to automate the design of complex systems. A greater level of experience with implementing different applications on a real DART-based system will guide future design choices for the ATaG runtime.

Software synthesis

In the context of the ATaG-based programming framework, software synthesis is the process of generating code for each node of the target sensor network deployment for the selected ATaG program. The code that is associated with each application-level functionality (abstract task) is to be provided by the programmer. The task of the software synthesis process is to generate the remainder of the software that is responsible for coordination and communication between the abstract tasks. To ease the task of software synthesis, we designed DART such that a majority of the code base either is agnostic to the application level functionality or can be customized by means of a configuration file that is generated by the software synthesizer. As an example, approximately 3000 lines of Java code runs on each sensor node in the ATaG program for object tracking (Section 2.5.1), of which only 100 lines are actually provided by the application developer and the rest is comprised of DART code that is used essentially unchanged and some glue code that is generated by the software synthesizer. The newly generated glue code is only 15 lines of Java that basically embeds the declarative part of the ATaG program into the runtime system, along with a one-line configuration file for each node in the target network that provides some state information to govern the node's behavior during the simulation.

Outline

The core ideas of ATaG have been individually explored in different contexts in the parallel and distributed computing community. There are also other approaches to the problem of macroprogramming of sensor networks being explored in the sensor networking community. Some of these were discussed previously in this chapter. Chapter 2 presents the ATaG programming model in detail with a description of a syntax and semantics of ATaG program. A

set of programming idioms are also provided to illustrate the formulation of oft-cited behaviors in sensor networking as ATaG programs. The design of the DART runtime system is the subject of Chapter 3, which describes the service provided by each of the DART components, the interactions between the various components, and implementation notes. Chapters 2 and 3 also include a discussion of future research directions in the context of the programming model and the design of the runtime system, respectively. Chapter 4 presents the visual programming and software synthesis environment for ATaG. A brief primer on the Generic Modeling Environment [21] precedes the discussion of the various modeling paradigms that are provided to the application developer. A case study is included in Chapter 5 to illustrate the development of an application consisting of two behaviors—object tracking and environment monitoring—using this programming environment. We conclude in Chapter 6.

