SRI International

CSL Technical Report SRI-CSL-07-01 ● January 19, 2007

Sensor Coordination Using Active Dataspaces

Mohamed Abdelhafez
Steven Cheung

This research is sponsored by the National Science Foundation under grant number CNS-0434997. The views herein are those of the authors and do not necessarily reflect the views of the sponsoring agency.

Sensor Coordination Using Active Dataspaces

Mohamed Abdelhafez
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250
mohamed.hafez@ece.gatech.edu

Steven Cheung
Computer Science Laboratory
SRI International
Menlo Park, CA 94025
steven.cheung@sri.com

Abstract
To ease application development for wireless sensor networks, we have developed a high-level, data-centric programming model, called *active dataspace* (ADS), and a prototype implementation of ADS on the TinyOS platform. An ADS is an active data repository that supports associative data access operations. The ADS model is based on the tuple space coordination model used in parallel computing, and extends it to make it applicable for sensor networks. The key elements of ADS include a construct, called *virtual tuple*, that enables a sensor node to express its capability to generate on demand a specified type of tuple, in-network aggregation primitives to support resource-efficient data accesses, and timeout tags for tuples to facilitate handling sensor network dynamics. ADS provides a high-level, expressive programming abstraction for developing a variety of sensor network applications that are resource efficient, and tackle the "come and go" characteristic of sensor networks.

Keywords: Wireless sensor networks, programming, data centric, tuple space

Chapter 1

Introduction

Programming wireless sensor networks is challenging because of the severe resource constraints and the dynamic (or "come and go") nature of these networks. In particular, the nodes and communication links in sensor networks are significantly less reliable than their counterparts in conventional computer networks, because of sensor failure, depletion of the energy reserve, changes in the environment that affect communication links, and attacks against the network. Also, sensor nodes may perform duty cycling to conserve energy. To cope with these network changes, developers for sensor network applications often need to juggle many low-level sensor coordination tasks, such as monitoring and keeping track of the states of neighbor nodes, and activating sensor nodes when their services are needed.

To facilitate developing sensor network applications, we have developed a high-level, data-centric programming model, called *active dataspace (ADS)*, and an initial implementation of ADS on a typical sensor network platform, TinyOS. An ADS is an active data repository that provides associative memory operations for data access. Our work is based on the tuple space coordination model (briefly reviewed in Chapter 2), and extends it to make it applicable for sensor networks.

The ADS model is designed to tackle the dynamic nature of sensor networks. In our model, sensors use a shared-memory abstraction to communicate among themselves—-depositing tuples to an ADS, and retrieving them from an ADS by specifying values of (some of) the tuple fields. Sensor nodes do not need to know the identity of the other nodes in order to communicate with them. Moreover, ADS enables nodes that are active at different times, and nodes for which there is no stable end-to-end communication path between them, to collaborate. In addition to the basic tuple space operations, we have designed new constructs in ADS to support resource conservation and adaptation to network changes. These constructs include a special type of tuple called *virtual tuple*—representing the capability of a node to generate a specified type of tuple—to support on-demand tuple generation, timeout tags for tuples to enable automatic removal of stale data or tuples from nodes that are no longer reachable, and aggregation primitives to support in-network processing.

To date, most sensor network applications were developed using node-centric, event-driven programming models (e.g., nesC/TinyOS). Although these models are expressive and can develop very resource-efficient applications, they are difficult to use. Significant progress has been made to ease sensor network application development. For example, Hood [11] and abstract regions [10] have proposed high-level programming primitives to efficiently support in-network aggregation operations. Programming models specific to certain application classes, such as Cougar [12], TinyDB [8], and EnviroTrack [1], have been proposed. TinyDB presents a relational-database-based approach for constructing energy-efficient, periodic data collection applications. EnviroTrack presents a powerful programming abstraction to facilitate the construction of an important class of applications—object detection and tracking. Directed diffusion [6] and Agilla [3] represent earlier work that is the closest to ADS. Directed diffusion presents a data-centric routing approach. Moreover, it uses a reinforcement-based adaptation technique to select the best path for routing data. Directed diffusion and ADS have similarities; they are both data centric, and have the ability to match data producers and data consumers based on the attributes of the data involved. Agilla presents a tuple space-based programming model developed for handling node mobility. In Agilla, every node maintains a tuple space, and when two nodes are within communication range, their tuple spaces can merge to support communication. ADS differs from directed diffusion and Agilla in that it supports an uncoupling style of communication in which the data producers and data consumers do not need to be active at the same time, and stable end-to-end communication paths between the producers and the consumers are not required. In the macroprogramming approach, one specifies some high-level operations to perform. These high-level operations are compiled automatically into low-level tasks, which are assigned to sensor nodes. Regiment [9] and Kairos [5] are examples of this approach. A main difference that distinguishes our work from earlier work is its focus on a high-level, general-purpose programming model that can tackle the "come and go" nature of sensor networks.

The rest of this paper is structured as follows. Chapter 2 briefly describes the tuple space model on which our work is based. Chapter 3 presents the design and an initial implementation of ADS. Chapter 4 uses a few examples to illustrate sensor network programming using ADS. Chapter 5 discusses future directions.

Chapter 2

Tuple Space Model

The ADS model is inspired by the tuple space model. The tuple space coordination model for parallel programming was first proposed by Gelernter [4], and a large body of work has been performed on this model, including extensions of its primitives, implementations on various platforms, integration with different parallel programming paradigms, and a variety of application programming experiments. In the tuple space model, processes use four basic operations to interact with a tuple space, namely, *in, rd, out*, and *eval*. The *in* and *rd* operations are used to remove and to read data tuples from the tuple space, respectively. The *out* and *eval* operations are used to create data and "active" tuples, respectively. An active tuple corresponds to a new process, which executes a specified program and becomes a (passive) data tuple when it terminates.

For example, if a producer process P generates data for a consumer process C, P executes an *out* operation, such as, out("mag-reading", 3). Then process C uses an *in* operation, such as in("mag-reading", ?r), to remove the tuple. The variable r is instantiated with the value 3 when the *in* operation completes. A tuple matches an *in* operation if all its fields can be "unified" with the corresponding fields of the *in* operation. (Two fields can be unified if both are actuals/fixed and have the same value, or if one of them is a formal/variable and they are of the same type.) Figure 2.1 depicts the interaction between the processes and the tuple space.

The tuple space coordination model is particularly suited for sensor networks because it supports a time- and identity-uncoupling style of communication. The sender and the receiver of a tuple do not need to exist at the same time, and do not need to know each other's identities. For instance, a sensor S may publish its magnetometer readings by inserting them into a tuple space, and then go into hibernation. While S is inactive, another node may become active and retrieve sensor reading tuples from the tuple space and aggregate these readings for vehicle-

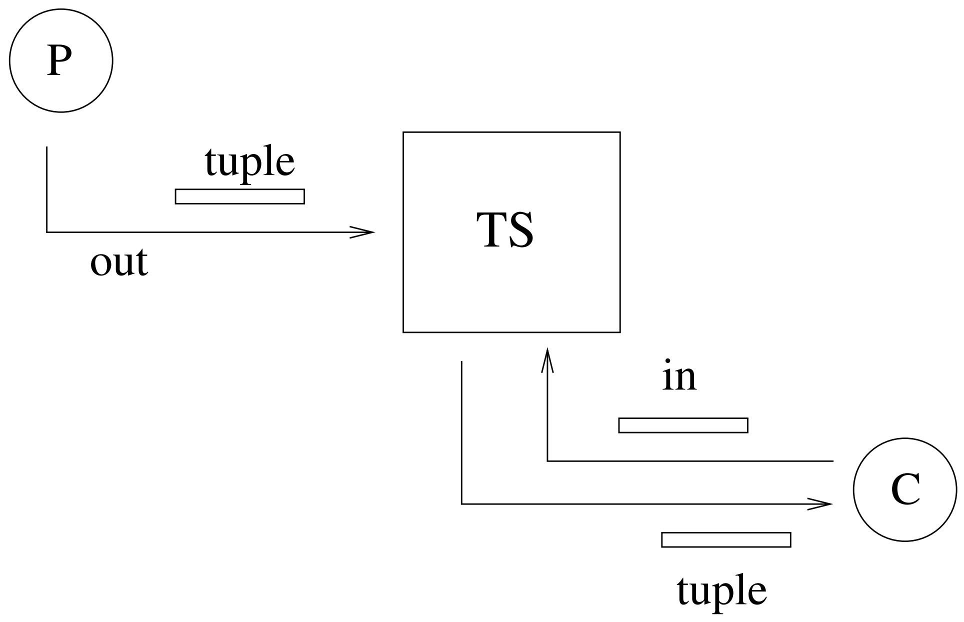


Figure 2.1: Components of the tuple space model and their interactions

tracking purposes. Moreover, in many sensor network applications, attributes like the time, location, and sensor values are the key information needed by the data consumer. On the other hand, the name of the sensor that produces a sensor report may be irrelevant. In the tuple space model, the same tuple can be deposited by any producer and then withdrawn by any consumer, enabling distributed sharing of data.

The tuple space model supports structured naming, where the tuple name is formed by the "actual" parameters in the operations, such as "mag-reading" in the above example. By specifying values for one or more fields, tuple space operations can generate or refer to tuples with the corresponding names. The data-centric communication aspect of the tuple space model simplifies the node-naming issues for sensor networks.

Another advantage of the uncoupling style of communication model is that it is applicable for intermittently connected networks (e.g., [2]) in which a stable end-to-end path between the sender and the receiver may not exist. In particular, a node can deposit data in a tuple space opportunistically. When a data consumer comes in contact with the tuple space, it may then retrieve the data tuple.

Chapter 3

ADS Model and Implementation

We present the main components of ADS and the operations used by the applications to interact with ADS. Like the tuple space model, ADS consists of two types of components—tuple space (TS) node(s) and ADS participant node(s). A TS node in ADS not only handles tuple storage and maintenance, but also provides aggregation functions. ADS participant node(s) deposit tuples into or retrieve tuples from the TS node. The novel features of ADS include operations that facilitate handling the dynamic nature of sensor networks and help in constructing resource-efficient sensor network applications.

3.1 Operations

The ADS model supports three main operations of the tuple space model, namely, *in*, *rd*, and *out*, and their nonblocking variants. In addition to these operations, the ADS model supports several new operations, namely, *outcap*, *interest*, *aggregate*, and *delete*. Briefly, *outcap* is used for specifying on-demand tuple generation capabilities, *interest* is a variant of *in* and *rd* for periodic data retrieval, *aggregate* is for in-network tuple aggregation, and *delete* is for removing "control" tuples used in *outcap* and *interest* operations.

3.1.1 in operation

The *in* operation retrieves a tuple and removes it from the TS. The prototype for this operation is as follows:

```
command result_t inop(tuple* tp, uint32_t** point, uint8_t tout)
```

where *tp* specifies the tuple to search for in the TS, *point* is an array of pointers to local variables that would be populated with the data from the retrieved tuple, and *tout* specifies the maximum amount of time to wait for a matching tuple.

If no matching tuple in the TS is found before the timeout, this operation terminates. We note that the *in* operation of the original tuple space model does not have the timeout feature. A process may wait indefinitely for the *in* operation to finish if there is no matching tuple. Because the communication channels in sensor networks may be unreliable and unpredictable, we added the timeout feature to ADS to ease sensor network programming. The *inp* operation is the nonblocking version of the *in* operation. If no matching tuple is found in the tuple space, the operation returns immediately.

3.1.2 rd operation

The *rd* and *rdp* operations are similar to the *in* and *inp* operations, except that they are used to *read* the values of the tuples without removing them from the tuple space.

3.1.3 out operation

The *out* operation is used to deposit tuples into the TS. The prototype for the *out* operation is as follows:

```
command result_t outop(tuple* tp, uint8_t tout)
```

3.1.4 outcap operation

The *outcap* operation supports on-demand tuple generation. When a node wants to advertize its tuple generation capabilities, it deposits a special type of tuple, called *virtual tuple*, into the TS. A virtual tuple differs from an (ordinary) data tuple in that it declares the capability of producing a specified type of data. The actual tuples are produced when there is a matching request from another node. Using virtual tuples, a producer node can stay idle until it receives a request for data generation, thus saving energy during the time intervals in which no consumer is interested in its data. This can be especially useful for event-driven data generation, for which the producer node does not know a priori when to generate the data. When the TS node receives an *in* operation that matches a virtual tuple, the request is forwarded to the node that deposited the virtual tuple. The producer node then generates the actual data and sends the corresponding tuple to the TS, which in turn forwards it to the requesting node. Figure 3.1 summarizes these steps.

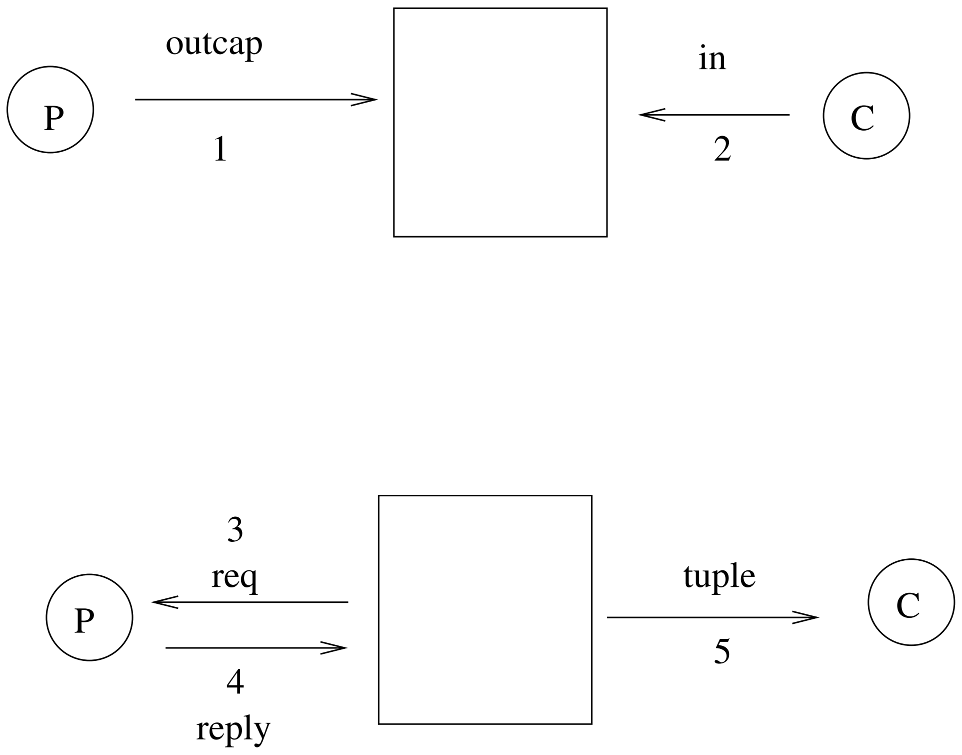


Figure 3.1: Outcap operation

3.1.5 interest operation

The *interest* operation retrieves tuples that match a specified criterion on a periodic basis. Using an interest operation, a consumer node can avoid continuously polling the TS for data retrieval. As a result, using this operation can reduce radio communication for the consumer, thus increasing its lifetime. In our ADS implementation, the TS stores the interests from various consumers. Whenever the TS receives a tuple, it tries to match the tuple with the stored interests. If the TS node finds a match, it forwards that tuple to the owner of that interest. If a virtual tuple matches an interest, the interest is forwarded to the owner of the virtual tuple. For periodic data retrieval, the interest specifies the period at which the consumer wants the data. The owner of the virtual tuple then schedules its production of the actual tuples based on the received interest, and performs *out* operations to send them to the TS, which in turn forwards them to the consumer node. The prototype for the interest command is as follows:

```
sendInterest(tuple* tp, uint32_t** point,
             uint8_t timeout, uint8_t period,
             uint8_t remove)
```

where *period* is used for periodic data retrieval, and *remove* is a flag indicating whether to remove the matching tuple from the TS, or keep a copy there for other nodes to retrieve.

3.1.6 aggregate operation

The *aggregate* operation is used to obtain aggregate information about a set of tuples that matches a specified criterion. For example, a set of nodes can find the one that has the largest node ID by depositing tuples that contain their IDs into the TS, and calling the aggregate operation to compute the maximum for the ID field among these tuples. As another example, the aggregate operation may be used to find the sum of sensor values from different producer nodes. In our design, the TS node performs the aggregate operations, and sends the aggregate results to the requesting nodes. This is especially useful when more than one consumer needs the aggregate information. Instead of replicating the data on all consumer nodes, the aggregate operation allows the consumers to request the aggregation result, while the tuples are stored only once in the TS node. Thus, using the aggregate operation may increase the lifetime of the consumer node, because of reduced radio communication and reduced processing.

3.1.7 delete operation

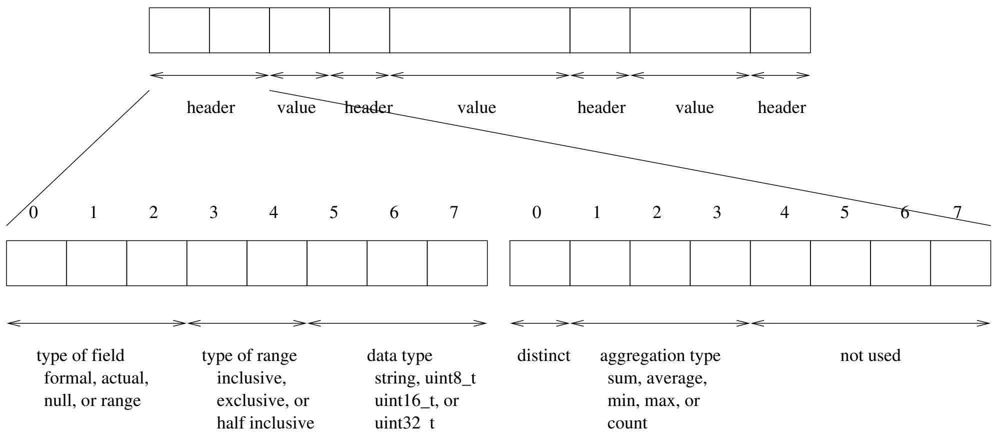
The *delete* operation is used to revoke virtual tuples or interests from the TS. This is useful when a node decides not to serve additional requests for data, for example, because of low battery, moving out of range, or a change in task assignment. This operation has two forms—*deleteTuples* and *deleteInterests*—which specify the tuples or interests to remove from the TS.

3.2 Tuple Representation

The basic (data) unit in the ADS model is a *tuple*. An ADS tuple is an array of fields, which may have different data types and sizes. Each field of the tuple is represented by a header and a value. (The value is omitted for formal or null fields.) As shown in Figure 3.2, the header subfield may be one or two bytes long, with the two-byte headers used for aggregation purposes. Moreover, the value subfield is of variable size, depending on the data type of the field.

Our ADS implementation provides methods for applications to initialize and add fields to tuples, which can then be used in ADS operations. The header for each field specifies whether that field is a *formal*, an *actual*, a *null*, or a *range*. For formal field arguments, one specifies a pointer to a variable that will be populated with the corresponding value of the matching tuple. Actual fields correspond to a fixed value, for example, a string "temp" or an integer 3. A null field represents "don't care" terms, and can match with any values. In a range field argument, one specifies the upper bound and the lower bound for the values that can match this field. The header also specifies the data type of the field, which may be a string or an integer of various sizes (uint8_t, uint16_t, and uint32_t). The string type is like C strings, which consist of an array of characters with a special terminating character. The three integer types occupy one, two, and four bytes of storage, respectively. The second byte of a header is used only for aggregation operations. The aggregation operations that are currently supported include sum, average, minimum, maximum, and count. The aggregation header also has a bit that specifies whether the matching tuples should have distinct values for that field.

In the current implementation, we limit the tuple size so that a tuple fits in a TinyOS packet. We made that choice to avoid the overheads associated with packet fragmentation and reassembly. A tuple is stored as an array of 20 bytes.



header
value
header
value
header
value
header
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
type of field formal, actual, null, or range
type of range inclusive, exclusive, or half inclusive
data type string, uint8_t uint16_t, or uint32_t
distinct
aggregation type sum, average, min, max, or count
not used

Figure 3.2: Tuple format (top) and field header format (bottom)

3.3 Tuple Storage

The TS node is responsible for tuple storage and processing. In the implementation, there are two different storage locations in the TS node; one is for tuples, and the other is for interests.

In the TS node, tuples are enclosed in a record that contains additional information including the timeout, which determines how long this tuple is valid, a unique ID for each record, the ID of the node that generated this tuple, and flags indicating whether the tuple is virtual, or whether it has been deleted.

An interest record is similar to a tuple record, except that it has two more fields, namely, the frequency and the last service time. The frequency determines how often this interest should be "activated". The last service time field is used to ensure fairness among multiple interests that match the same tuples. Whenever an interest matches a tuple, the last service time field is updated with the current time. When another tuple arrives that matches the same interest, the last service time and frequency fields are checked to make sure that this interest does not receive more tuples than it needs, and also that interests have a fair chance of being served. The interest record has additional flags; the persistant flag determines if the interest is to remain active after being served, and the remove flag determines if the tuple used to serve that interest should be removed from the TS.

3.4 Message Type

There are six ADS message types.

- *OperationMessage*, which is for sending ADS operations from ADS participants to the TS node. OperationMessage has three fields—tuple, type of operation, and timeout. Timeout specifies the expiration time for the operation.

- *RequestMessage*, which is used by the TS node to obtain tuples from the producer node when a virtual tuple matches a tuple request. RequestMessage has three fields—tuple, timeout, and frequency. The latter two fields are used when the request pertains to periodic data retrieval.

- *ReplyMessage*, which is used by a producer to reply to a RequestMessage request. This message has two fields—tuple and timeout.

- *TSReply*, which is used by the TS node to send a tuple requested by a consumer node. This message has one field—tuple.

- *InterestMessage*, which is used by a consumer to express an interest to obtain a specified tuple. The interest message has four fields—tuple, remove flag, frequency, and timeout. Remove flag indicates if the matching tuple should be removed from the TS. Frequency specifies the period for periodic tuple requests. Timeout specifies the expiration time for the interest.

- *AggregateMessage*, which is used by a node to obtain aggregate information about a specified set of tuples. AggregateMessage has four fields—tuple, which specifies the criteria for the matching tuples and the type of aggregation to perform; min and max, specifying the minimum and maximum number of matching tuples needed for the request; and remove flag, which indicates whether the matching tuples should be removed from the TS after the operation.

Chapter 4

Examples

We present examples of developing sensor network applications using ADS. The first example concerns controller nodes sending commands to a set of worker nodes. It demonstrates a use of the *interest* operation. The second example involves on-demand data generation. It demonstrates how *virtual tuples* may facilitate that kind of application. The third example is about object localization. It demonstrates using the *aggregate* operation for in-network aggregation.

4.1 Controller-Worker Pattern

In this example, one or more controllers send commands to a set of worker (sensor) nodes in the network. The workers interpret the commands and perform the corresponding actions. This sensor coordination pattern may occur in dynamic task assignments. A controller may be the base station or a cluster head that manages a set of nodes in its region. Worker nodes may have different capabilities. For example, they may be equipped with different sensor suites.

In addition to the basic TS operations, such as *out*, this example demonstrates a use of the *interest* operation. Specifically, the nodes that are capable of handling commands may deposit interest tuples into the TS. The controllers would send "command" tuples to the TS, which would match them with the interests and then forward them to the corresponding workers. An example command tuple is {ControllerID, SensorID, Command_code}. The controller IDs and sensor IDs may be node IDs, or an attribute (e.g., location) that would identify a node or a set of nodes. The command code specifies the command to be carried out.

To put an interest into the TS, the nesC code for the worker nodes includes the following lines:

```
ClientTuple ct;

call ADS.initTuple(&ct);
call ADS.addValueField(&ct, BYTE | FORMAL, &controller_id, 0);
call ADS.addValueField(&ct, BYTE, &moteID, 0);
call ADS.addValueField(&ct, BYTE | FORMAL, &command_code, 0);

call ADS.sendInterest(&ct.tp, ct.pointers, TIMEOUT, PERIOD, 0);
```

A worker node first initializes the interest tuple and then adds three fields to the interest tuple. The first field is the ID of the controller node. Note that the sensor does not know which controller node will send the command. Instead, it requests the controller ID and specifies a local variable "controller_id" for storing it. The word "FORMAL" indicates that the field pertains to a variable. The second field is the sensor ID, which is the mote ID of the node. The third field is the command code. The sendInterest operation is used to send the interest tuple, which has the following arguments: the pointers to the local variables to hold the returned values, a timeout for this interest, a period specifying how often the node can handle commands, and a flag indicating whether the matching tuple should be removed from the TS. The interest would match any tuple that has three single-byte fields with the second field equaling the mote ID or being a null field.

When the TS receives a command tuple that matches an interest, it forwards the tuple to the worker node that deposited the interest. The worker node is notified by means of an event to process the tuple and handle the commands received, say, using a switch statement on the "command_code" variable, which is populated with the proper value when that event is fired.

To generate and send a command tuple, the code for the controller node includes the following lines:

```
ClientTuple ct;

call ADS.initTuple(&ct);
call ADS.addValueField(&ct, BYTE, &moteID, 0);
call ADS.addRangeField(&ct, BYTE, 10, 15, 0, 0);
call ADS.addValueField(&ct, BYTE, &command_code, 0);

call ADS.out(&ct.tp, TIMEOUT);
```

A controller node initializes the command tuple with three fields: controllerID, which is set to its mote ID; sensor ID, which is used for specifying the target worker nodes and is a range from 10 to 15 in this example; and command code, which is stored in the variable "command_code". For example, if the controller ID is 50 and the command code is 24, the command tuple is {50, [10-15], 24}.

4.2 On-demand Data Generation

Depending on the application, a sensing and reporting task can be periodic or event driven. In the periodic case, it can be realized by having the producer nodes deposit tuples into the TS using *out* operations; the consumer nodes retrieve these tuples using *in* or *rd* operations. If the sensor data is needed only under some conditions or when certain events occur, it may be more efficient for the producers to use virtual tuples, which represents the capability to generate the corresponding tuples. When a consumer sends a request that matches the virtual tuple, the TS forwards the request to the producer node, which then generates the actual data tuples. This example concerns using ADS to support on-demand data retrieval applications.

To put a virtual tuple into the TS, a producer may perform the following operations:

```
ClientTuple ct;

call ADS.initTuple(&ct);
call ADS.addValueField(&ct, BYTE, &moteID, 0);
call ADS.addValueField(&ct, SHORT, &sensor_data, 0);

call ADS.outcap(&ct.tp, TIMEOUT);
```

In this code segment, the producer creates a virtual tuple containing two fields, namely, producer ID and sensor reading. Then the producer uses the outcap operation to deposit the virtual tuple into the TS. Note that the code pertaining to tuple initialization and field addition is similar to that in the previous example. The main difference is the use of the *outcap* operation instead of the *out* operation. All ADS operations share the same tuple structure and follow similar steps for tuple generation and population. This design simplifies the programming model, and thus makes it easier to develop applications.

To retrieve the data, a consumer may perform the following operations:

```
ClientTuple ct;

call ADS.initTuple(&ct);
call ADS.addNullField(&ct, BYTE, 0);
call ADS.addValueField(&ct, SHORT | FORMAL, &sensor_data, 0);

call ADS.in(&ct.tp, TIMEOUT);
```

The consumer creates a request tuple whose first field can match any (byte) value and whose second field contains a pointer argument for storing the result. When the TS receives the request tuple, it searches through its database. If the TS finds a matching virtual tuple, it forwards the request to the producer node corresponding to the virtual tuple. Moreover, when the producer receives the request from the TS, the *requestReceived* event fires.

The following code segment shows an example handler for the *requestReceived* event. The local variable "sensor_data" contains the current sensor reading. The generated tuple is sent to the TS using the reply operation.

```
ClientTuple ct;

call ADS.initTuple(&ct);
call ADS.addValueField(&ct, BYTE, &moteID, 0);
call ADS.addValueField(&ct, SHORT, &sensor_data, 0);

call ADS.reply(&ct.tp, TIMEOUT);
```

4.3 Target Localization

Determining the location of target objects is a common task for object detection and tracking. In this example, several sensor nodes report their readings, and a node computes the location of the target based on these readings. The computation used to calculate the (x, y)-coordinates of the target, derived from [10], is summarized as follows:

$$c_x = \frac{\sum_i R_i x_i}{\sum_i R_i}$$

$$c_y = \frac{\sum_i R_i y_i}{\sum_i R_i}$$

where R_i is the sensor reading from node i, x_i and y_i are the (x, y)-coordinates of node i, and c_x and c_y are the computed (x, y)-coordinates of the target.

After a producer node gets a sensor reading, it multiplies the value with its (x, y)-coordinates, and sends a tuple containing these values, a string "Loc", and its mote ID to the TS. The following excerpt is from an implementation for the producer nodes.

```
event result_t SensorADC.dataReady(uint16_t data) {
   atomic
   {
        reading = data;
        xreading = x * reading;
        yreading = y * reading;

        post reportTask();
   }
   return SUCCESS;
}

task void reportTask() {
 //prepare tuple and send it
  clientTuple ct1;

  call ADS.initTuple(&ct1);
  call ADS.addValueField(&ct1, STRING, "Loc", 0);
  call ADS.addValueField(&ct1, BYTE, &moteID, 0);
  call ADS.addValueField(&ct1, SHORT, &reading, 0);
  call ADS.addValueField(&ct1, SHORT, &xreading, 0);
  call ADS.addValueField(&ct1, SHORT, &yreading, 0);

  call ADS.outop(&ct1.tp, TIMEOUT);
}
```

The consumer node uses the aggregate operation to obtain the sum of the sensor readings and the x- and y-coordinate-weighted readings to calculate the centroid. Specifically, the consumer requests aggregate information for a set of tuples such that the first field contains a string "Loc", and the second field (mote IDs) has distinct values. As an example of using the range construct, the consumer also indicates that the aggregate computation requires at least four tuples and a maximum of six tuples. When the TS node sends the aggregate data to the consumer, the event *tupleProcessed* is fired and the variables (corresponding to the aggregate sensor readings) are populated with the received results. The consumer then calculates the centroid of the readings. Finally, it calls a command (not shown) to report the calculated location, say, to the base station.

```
task void getTask()
{
   clientTuple ct1;

   call ADS.initTuple(&ct1);

   call ADS.addValueField(&ct1, STRING, "Loc", 0);
   call ADS.addNullField (&ct1, BYTE, DISTINCT);
   call ADS.addValueField(&ct1, SHORT | FORMAL_MASK, &readings, SUM);
   call ADS.addValueField(&ct1, SHORT | FORMAL_MASK, &xreadings, SUM);
   call ADS.addValueField(&ct1, SHORT | FORMAL_MASK, &yreadings, SUM);

   call ADS.aggregate(&ct1.tp, ct1.pointers, 4, 6, 0);
}


event result_t ADS.tupleProcessed()
{
 centroidx = xreadings / readings;
 centroidy = yreadings / readings;

  post sendTask();

  return SUCCESS;
}
```

We have implemented this example using our ADS prototype, and found that the application on consumer node(s) requires 12436 bytes in ROM and 555 bytes in RAM. The application on producer node(s) requires 13444 bytes in ROM and 552 bytes in RAM.

We ran experiments using the TinyOS Simulator (TOSSIM) [7]. The experimental setup includes four producer nodes, one consumer node, and one TS node. The producer nodes send their readings to the TS once every five seconds. Moreover, the sensor reading tuples have a timeout of five seconds. The consumer node requests the aggregate values of the tuples from the TS once every five seconds. In terms of message costs per cycle, this results in one message for an out operation from each producer to the TS node, one message from the consumer for the aggregation request, and one message from the TS node to the consumer for the result of the aggregation. In other words, six messages are exchanged in every cycle. Without using the aggregate construct, the number of messages would be nine per cycle.

To evaluate the overhead of ADS, we implemented a version of the same application directly in nesC. For the nesC version, the application for consumers requires 10018 bytes in ROM and 324 bytes in RAM, while the application for producers requires 12098 bytes in ROM and 356 bytes in RAM. In the nesC version, the producer nodes send the data directly to the consumer once every five seconds, and the consumer then computes and broadcasts the result. In this version, the number of messages is four per cycle.

For this application, the memory overhead for using ADS is about 10-20% increase in ROM and 50-70% increase in RAM. Moreover, the number of messages is four for the nesC version and six for the ADS version. Also, the ADS version uses an extra node to act as the TS node. We note that the ADS version can readily be extended to incorporate duty cycling to reduce energy consumption (see Chapter 5), and it is nontrivial to adapt the nesC version to perform duty cycling.

Chapter 5

Future Directions

We discuss ongoing work and possible extensions to the current ADS implementation.

5.1 Power Management

Power management is important in sensor network applications because typically sensor nodes have limited energy reserves, and it may be difficult to perform battery replacement or recharge, depending on the application environment. The current implementation of ADS supports simple power management by means of virtual and interest tuples. This is achieved by having sensor nodes deposit these tuples into the TS and then going into a low-energy mode. The TS node "wakes up" the nodes by forwarding matching requests or tuples to them. The current implementation works only on the Mica2 and Mica2dot platforms, because it uses the duty cycling commands of the $CC1000RadioIntM$ module.

For applications that involve periodic data generation, the consumer node could send an interest that indicates the frequency at which data is needed. This interest, when matched with a virtual tuple from a producer, is forwarded to that producer. The consumer and the producer then synchronize their duty cycling schedule based on the requested data generation frequency.

For applications that involve dynamic data collection or event-driven data generation, we need a different approach because no predetermined schedule can be used. In that case, the producer can generate a virtual tuple declaring its ability to produce a specified type of tuple before entering the low-energy mode. When the TS node receives a request for that tuple, it wakes up and passes the request to the producer node. The same approach is applicable for consumer nodes, which may deposit an interest into the TS, and wait for a matching tuple while in a low-energy mode.

In our implementation, a node that performs duty cycling checks for incoming messages at a slow rate. We need a reliable means to send messages to nodes in the low-energy mode. To this end, we use the *WakeupComm* module, in the Drip package of TinyOS, for providing reliable communication. The *WakeupComm* module has an interface similar to that of *GenericComm*, the commonly used TinyOS communication module. Internally, *WakeupComm* uses a timer to repeatedly send the same message multiple times. Two main issues arise from the use of this module. First, retransmissions are expensive and, therefore, should be avoided whenever possible. Second, it cannot handle multiple concurrent message transmissions. The first issue can be addressed by incorporating an acknowledgment mechanism. When the TS node receives an acknowledgment, it signals the *WakeupComm* module to stop retransmitting the message. We plan to address the first issue by using link layer acknowledgments in TinyOS, which indicates the presence of acknowledgments by using a flag in the sendDone event of the *WakeupComm* module. A cost-effective solution to address the second issue is a task for future work.

5.2 Clustering

The current ADS implementation assumes that all nodes are within the radio range of the TS node. This design is not scalable. For large sensor networks, we may deploy the concept of clustering, with each cluster containing a TS node that can communicate with all the nodes in the cluster. Intercluster communication can be performed using a hierarchical scheme, with the TS node of the top cluster of each subtree carrying summarized information about the tuples contained in its descendant clusters. If a consumer node requests a tuple from its own TS node, and no matching tuple is found, the TS node forwards the request to its parent cluster. This process is recursively performed until a matching tuple is found or the root cluster is reached.

Bibliography

[1] T. F. Abdelzaher, B. M. Blum, Q. Cao, D. Evans, J. George, S. George, T. He, L. Luo, S. H. Son, R. Stoleru, J. A. Stankovic, and A. Wood. EnviroTrack: An environmental programming model for tracking applications in distributed sensor networks. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, pages 582–589, Mar. 2004.

[2] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss. Delay-tolerant networking: An approach to interplanetary Internet. *IEEE Communications Magazine*, pages 128–136, June 2003.

[3] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662, Columbus, Ohio, June 2005.

[4] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

[5] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[6] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the* 6^{th} *Annual International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 56–67, Boston, MA, Aug. 2000.

[7] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys'03)*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[8] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the ACM SIGMOD Conference*, San Diego, CA, June 2003.

[9] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the First International Workshop on Data Management for Sensor Networks*, Toronto, Canada, Aug. 2004.

[10] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of the* 1^{st} *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, Mar. 2004.

[11] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, June 2004.

[12] Y. Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *Sigmod Record*, 31(3), Sept. 2002.