# Macroprogramming Spatio-temporal Event Detection and Data Collection in Wireless Sensor Networks: An Implementation and Evaluation Study

Hiroshi Wada, Pruet Boonma and Junichi Suzuki
Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125-3393
{shu, pruet, jxs}@cs.umb.edu

## Abstract

*This paper proposes and evaluates a spatio-temporal macroprogramming paradigm for wireless sensor networks (WSNs). The proposed paradigm, called SpaceTime Oriented Programming (STOP), is designed to reduce the complexity to program event detection and data collection by specifying them from a global viewpoint as a whole rather than a viewpoint of sensor nodes as individuals. STOP treats space and time as first-class citizens and combines them as spacetime continuum. A spacetime is a three dimensional object that consists of a two spatial dimensions and a time playing the role of the third dimension. STOP allows application developers to program event detection and data collection to spacetime, and abstracts away the details in WSNs. The notion of spacetime provides an integrated abstraction to seamlessly express event detection and data collection for both the past and future in arbitrary spatio-temporal resolutions. This paper describes the implementation of STOP, and evaluates the performance of applications built with STOP.*

## 1. Introduction

Macroprogramming is an emerging programming paradigm for wireless sensor networks (WSNs). It allows developers to implement each WSN application from a global viewpoint as a whole rather than a viewpoint of sensor nodes as individuals. A macro-program specifies an application's global behavior. It is transformed to node-level (micro) programs, and the micro-programs are deployed on individual nodes. Macroprogramming aims to increase the simplicity and productivity in WSN application programming.

WSN applications are traditionally classified into event detection and data collection applications. In an event detection application, nodes send sensor data to base stations only when they detect an event (e.g., significant changes in their sensor readings). In a data collection application, individual nodes periodically send sensor data to base stations. Alternatively, central entities (e.g., servers and base stations) query nodes to collect sensor data. Existing macroprogramming languages consider event detection and data collection largely in isolation, as discussed in Section 7. However, WSN applications are often required to implement both aspects when they are designed to detect and monitor *spatiotemporally dynamic events*, which change over space and time (e.g., wild fires, oil spills and chemical/gas dispersions). For example, once detecting an event, an application may be required to collect past sensor data in the event area to seek any foretastes that help understand the nature of the event.

This paper proposes a new macroprogramming paradigm, called SpaceTime Oriented Programming (STOP). STOP provides an abstraction to program spatio-temporal event detection and data collection as the global behaviors of each WSN application. STOP considers both spatial and temporal aspects of sensor data, and treats space and time as first-class citizens in programming. Space and time are combined as *spacetime* continuum. A spacetime is a three dimensional object that consists of a two spatial dimensions and a time playing the role of the third dimension. STOP allows developers to program event detection and data collection *to spacetime*, and abstracts away the low-level details in WSNs, such as how many nodes are deployed, how nodes are connected and synchronized, and how packets are routed. The notion of spacetime provides an integrated abstraction to seamlessly express event detection and data collection for both the past and future in arbitrary spatio-temporal resolutions.

Moreover, STOP provides several mechanisms to customize micro-programs. The customization mechanisms allow developers to flexibly tune the performance and resource consumption of their applications by altering the details in event detection and data collection algorithms.

This paper is organized as follows. Section 2 overviews a motivating application that STOP is currently implemented and evaluated for. Section 3 describes how the STOP macroprogramming language is designed and how the language is used to specify event detection and data collection. Section 4 presents how macro-programs work on the STOP runtime environment and how the runtime environment is implemented. Section 5 describes how to customize micro-programs. Section 6 shows simulation results to characterize the perfor-

1530-1605/08 $25.00 © 2008 IEEE

mance of applications built with STOP. Sections 7 and 8 conclude with some discussion on related work and future work.

## 2. A Motivating WSN Application

STOP is designed to implement spatio-temporal event detection and data collection in WSNs, each of which consists of battery-operated stationary sensor nodes and several base stations. As an example application, STOP is currently used and tested for coastal oil spill detection and monitoring.

Oil spills occur frequently[1] and have enormous impacts on maritime/on-land businesses, nearby residents and the environment. When an oil spill occurs due to, for example, broken equipment of a vessel and coastal oil station, illegal oil dumping or terrorism, an in-situ WSN of fixed buoy-attached sensor nodes[2] (e.g., fluorometers[3] ) detects and monitors the spill. Oil may move and spread, change the direction of movement, and split into multiple chunks. Some chunks may burn, and others may evaporate and generate toxic fumes.

An in-situ WSN is expected to provide real-time sensor data to human operators so that they can efficiently dispatch first responders to contain spilled oil in the right place at the right time and avoid secondary disasters by directing nearby ships to evacuate, alerting nearby facilities or evacuating people from nearby beaches. In-situ WSNs can quickly deliver more accurate information (sensor data) to operators than visual observation from the air or coast. Also, in-situ WSNs are more operational and less expensive than radar observation with aircrafts or satellites [5]. (In-situ WSNs can operate during nighttime and poor weather, which degrade the quality of airborne and satellite observation.)

## 3. STOP Macroprogramming Language

STOP addresses the following requirements for the design of its macroprogramming language.

**Conciseness.** The conciseness of programs increases the ease of writing and understanding. This can improve the productivity of application development.

**Extensibility.** Extensibility allows application developers to introduce their own (i.e., user-defined) operators for in-network processing for meeting their application needs.

**Seamless integration of event detection and data collection.** As described in Section 1, STOP is required to support both event detection and data collection to implement applications for spatiotemporally dynamic events.

In order to satisfy the above requirements, the STOP macroprogramming language is designed as an extension to Ruby[4]. Ruby is an object oriented language that supports dynamic typing. The notion of objects combines program states and functions, and modularizes the dependencies between states and functions (i.e., which functions are supposed to change which states). This simplifies programs and improves their readability [6]. Dynamic typing makes programs concise and readable by omitting type declarations and type casts [7]. This allows application developers to focus on their macroprogramming logic without considering type-related housekeeping operations.

In general, a programming language can substantially improve its expressiveness and ease of use by supporting domain specific concepts inherently in its syntax and semantics [8]. To this end, the STOP macroprogramming language is defined as an embedded domain specific language (DSL) of Ruby. Ruby accepts embedded DSLs, which extend Ruby's constructs with particular domain specific constructs. By leveraging this mechanism, the STOP macroprogramming language reuses Ruby's syntax/semantics and introduces new keywords and primitives specific to spatio-temporal event detection and data collection (e.g., time, space, spacetime and spatio-temporal resolutions). Moreover, STOP assumes the JRuby interpreter[5] to execute macro-programs so that they can reuse existing Java libraries.

Ruby can encapsulate a code block as a *process object* and accept a user-defined operator as a process object. With this mechanism, the STOP macroprogramming language allows developers to define new user-defined operators to meet their application needs.

In macroprogramming, developers need to write an event detection and a corresponding handler to respond to the event. They also need to write a data collection and a corresponding hander to process collected data. Ruby supports *closures*, each of which modularizes a code block as an object (similar to an anonymous method). The STOP macroprogramming language uses closures to define handlers and concisely associate them with event detection or data collection.

The STOP macroprogramming language supports *proactive* (one time or periodical) and *reactive* (or event-driven) data collection. An reactive data collection is specified as a combination of event detection and data collection.

### 3.1. Proactive Data Collection

Proactive data collection is executed one time or periodically. It is used to implement data collection applications. Proactive data collection pairs a *data query* and a corresponding *data handler* to process obtained data. Listing 1 shows a STOP macro-program that specifies several proactive data collections. This program is visualized in Figure 1.

A spacetime is created at Line 5. In STOP, a class is instantiated with the `new()` class method. This spacetime (`sp`) is defined as a polygonal prism consisting of a triangular space (`s`) and a time period of an hour (`p`). STOP supports the concepts of absolute time and relative time, and allows application developers to denote relative time as a number annotated

[1] The US Coast Guard reports that 50 oil spills occurred in the US shores in 2004 [1], and the Associated Press reported that, on average, there was an oil spill caused by the US Navy every two days from 1990 to 1997 [2].

[2] Each sensor node is installed in a sealed and waterproof container.

[3] Fluorescence is a strong indication of the presence of oils. Certain compounds in oil absorb ultraviolet light, become electronically excited and fluoresce [3]. Different types of oil yield different fluorescent intensities [4].

[4] www.ruby-lang.org

[5] jruby.sourceforge.net

with its unit (`Week`, `Day`, `Hr`, `Min` or `Sec`) (Line 4).

`get_space_at()` is a method that the `Spacetime` class has (Table 1). It is called on a spacetime to obtain a snapshot space at a given time in a certain spatial resolution. In Line 7, an obtained space, `s1`, contains sensor data available on at least 60% of nodes (the third parameter) in the space at 30 minutes before (the first parameter) with a 20 seconds time band (the second parameter).

**Listing 1:** A STOP Macro-Program for an Proactive Data Collection

```
1  points = [ Point.new( 10, 10 ),
2    Point.new( 100, 100 ), Point.new( 80, 30 ) ]
3  s = Polygon.new( points )
4  p = RelativePeriod.new( NOW, Hr -1 )
5  sp = Spacetime.new( s, p )
6
7  s1 = sp.get_space_at( Min -30, Sec 20, 60 )
8  avg_value = s1.get_data( 'f-spectrum', AVG, Min 3 ) {
9    | data_type, value, space, time |
10   # the body of a data handler comes here. }
11
12 spaces = sp.get_spaces_every( Min 5, Sec 10, 80 )
13 max_values = spaces.collect { |space|
14   space.get_data( 'f-spectrum', MAX, Min 2 ){
15     | data_type, value, space, time |
16     # data handler
17     if value > avg_value then ...  }}
18
19 name = 'f-spectrum'
20 event_spaces =
21  spaces.select{|s| s.get_data(name, STDEV, Min 5)<=10)}
22      .select{|s|
23        s.get_data(name, AVG, Min 5) -
24        spaces.prev_of(s).get_data(name, AVG, Min 5)>20)} }
```

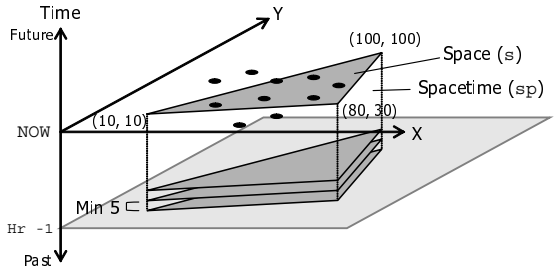



**Figure 1:** An Example Proactive Data Collection

`get_data()` is used to specify a data query. It is called on a space to collect sensor data available on the space (the first parameter) and process the collected data with a given operator (the second parameter). STOP provides several data aggregation operators shown in Table 2. In Line 8, `get_data()` obtains the average of fluorescence spectrum (`'f-spectrum'`) data in the space `s1`. The third parameter of `get_data()` specifies the tolerable delay (i.e., deadline) to collect and process data (three minutes in this example).

`get_data()` can accept a data handler as a closure that takes four parameters: the type of collected data, the value of collected data, the space where the data is collected, and the time when the data is collected. In this example, a code block from Line 9 to 10 is a closure, and its parameters contain a string `'f-spectrum'`, the average fluorescence spectrum in `s1`, the space `s1` and the time instant at 30 minutes before. An arbitrary data handler can be written with these parameters.

`get_spaces_every()` is called on a spacetime to obtain a discrete set of spaces that meet a certain spatio-temporal resolution. In Line 12, this method returns spaces at every five minutes with the 10 seconds time band, and each space contains data available on at least 80% of nodes within the space. Then, the maximum data is collected from each space (Lines 13 and 14). In STOP, a list has the `collect()` method[6], which takes a closure as its parameter, and the closure is executed on each element in a list. In this example, each element in `spaces` is passed to the `space` parameter of a closure (Line 15).

`select()` is used to obtain a subset of a list based on a certain condition specified in a closure. From Line 20 to 24, `event_spaces` obtains subset spaces, each of which yields 10 or lower standard deviation of data and finds a 20 or higher degrees difference in average data compared with a previous space at five minutes before.

**Table 1:** Key Methods in STOP

| Method | Description |
|---|---|
| Spacetime::get_space_at | Returns a snapshot space at a given time |
| Spacetime::get_spaces_every | Returns a set of snapshot spaces |
| Spacetime +/- spacetime | Returns an union/difference of two spacetimes |
| Space::get_data | Executes a data query |
| Space::get_node | Returns a node in a space |
| Space::get_nodes | Returns a set of nodes in a space |
| Space::get_border | Returns a set of nodes that form the border of a space |
| Space +/- Space | Returns an union/difference of two spaces |

**Table 2:** Data Aggregation Operators in STOP

| Operator | Description |
|---|---|
| COUNT | Returns the number of collected data |
| MAX | Returns the maxmum value among collected data |
| MIN | Returns the minimum value among collected data |
| SUM | Returns the summation of collected data |
| AVG | Returns the average of collected data |
| STDEV | Returns the standard deviation of collected data |
| VAR | Returns the variance of collected data |

### 3.2. Reactive Data Collection

Reactive (or event-driven) data collection is used to implement event detection applications. It is performed when an event is detected. It pairs an *event specification* and a corresponding *event handler* to respond to the event. Listing 2 shows a STOP macro-program that specifies an reactive data collection. It is visualized in Figure 2. This example assumes that each node detects an event (oil spill) when fluorescence spectrum exceeds 300nm. Once an oil spill is detected, this macro-program collects sensor data from an event area in the last 30 minutes and examines the source of the oil spill. This macro-program also collects sensor data from the event area over the next one hour in a high spatio-temporal resolution. The data is used to monitor how the detected oil spreads.

[6]In Ruby, a method can be called without parentheses when it takes no parameters.

Reactive data collection is specified with GLOBALSPACE, a special space that covers the whole observation area. An event specification is defined as a closure of GLOBALSPACE's select() method (Line 1). In this example, when sensor data exceeds 300nm in some area, select() returns the space. The on_event() method is called on the space.

An event handler is specified as a closure of on_event() (Line 4 to 23). Its parameters are the type of an event, the value of an event, the space where an event occurred, and the time when an event occurred (Line 5). Developers can use these parameters to write event handlers. In Line 9, a spacetime (sp1) is created to cover event_space and a period of 30 minutes. Then, sp1 is used to examine how many nodes observed 280nm or higher fluorescence spectrum every six minutes. In Line 18, another spacetime (sp2) is created to specify a future data collection. sp2 covers a larger space than event_space for an hour in the future. It is used to monitor the maximum fluorescence spectrum every three minutes.

**Listing 2:** A STOP Macro-Program for an Reactive Data Collection

```
1  GLOBALSPACE.select{
2    # an event specification
3    |s| s.get_data('f-spectrum', MAX) > 300 }
4    .on_event{
5      | event_type, value, event_space, event_time |
6      # event handler
7
8      # query for the past
9      sp1 = Spacetime.new(event_space,event_time,Min -30)
10     past_spaces = sp1.get_spaces_every(Min 6,Sec 20,50)
11     num_of_nodes =
12       past_spaces.get_nodes.select{|node|
13         node.get_data('f-spectrum',Min 3) > 280}.size
14
15     # query for the future
16     s2 = Circle.new(
17       event_area.centroid, event_area.radius * 2 )
18     sp2 = Spacetime.new( s2, event_time, Hr 1 )
19     future_spaces =
20       sp2.get_spaces_every( Min 3, Sec 10, 80 )
21     future_spaces.get_data( 'f-spectrum', MAX, Min 1 ){
22       | data_type, value, space, time |
23       # data handler } } }
```

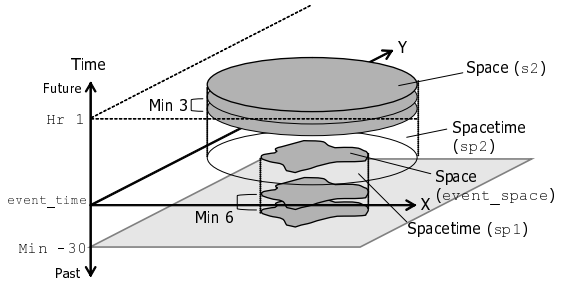



**Figure 2:** An Example Reactive Data Collection

### 3.3. User-defined Operators

The STOP macroprogramming language allows application developers to introduce their own (i.e., user-defined) operators in addition to the predefined operators shown in Table 2. In STOP, both predefined and user-defined operators are implemented in the same way.

Listing 3 shows the implementations of SUM, COUNT and AVG operators (Table 2). Each operator is defined as a *process object*, which is a code block that can be executed with the call() method. (See Line 14 for an example.) The keyword proc declares a process object, and its implementation is enclosed between the keywords do and end. sensor_readings is an input parameter to each operator (i.e., a set of sensor data to process) (Lines 1, 9 and 13).

Listing 4 shows an example user-defined operator, CENTROID, which returns the centroid of sensor data. This way, developers can define and use arbitrary operators that they need in their applications

**Listing 3:** Implementation of Predefined Operators

```
1  SUM = proc do |sensor_readings|
2    sum = 0.0
3    sensor_readings.each do |sensor_reading|
4      sum += sensor_reading.value
5    end
6    sum
7  end
8
9  COUNT = proc do |sensor_readings|
10   sensor_readings.size
11 end
12
13 AVG = proc do |sensor_readings|
14   SUM.call(sensor_readings)/COUNT.call(sensor_readings)
15 end
```

**Listing 4:** An Example User-defined Operator

```
1  CENTROID = proc do |sensor_readings|
2   centroid = [0, 0] # indicates a coordinate (x, y)
3   sensor_readings.each do |sensor_reading|
4    centroid[0] += sensor_reading.value*sensor_reading.x
5    centroid[1] += sensor_reading.value*sensor_reading.y
6   end
7   centroid.map{ |value| value / sensor_readings.size }
8  end
```

## 4. STOP Implementation

STOP is currently implemented with an application architecture that leverages mobile agents in a push and pull hybrid manner (Figure 3). In this architecture, each WSN application is designed as a collection of mobile agents, and there are two types of agents: *event agents* and *query agents*. An event agent (EA) is deployed on each node. It reads a sensor at every duty cycle and stores its sensor data in a data storage on the local node. When an EA detects an event (i.e., a significant change in its sensor data), it replicates itself, and a replicated agent carries (or pushes) sensor data to a base station by moving in the network on a hop-by-hop basis. Query agents (QAs) are deployed at Agent Repository (Figure 3), and move to a certain spatial region (a certain set of nodes) to collect (or pull) sensor data that meet a certain temporal range. When EAs and QAs arrive at the STOP server, it extracts the sensor data the agents carry, and stores the data to a spatio-temporal database (STDB).

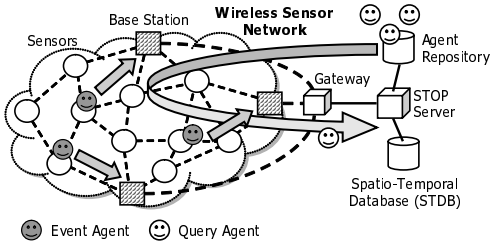



**Figure 3:** A Sample WSN Organization

At the beginning of a WSN operation, the STOP server examines network topology and measures the latency of each link by propagating a measurement message (similar to a hello message). EAs and QAs collect topology and latency information as moving to base stations. When they arrive at the STOP server, they update the topology and latency information that the STOP server maintains. The STOP server also maintains each node's physical location through a certain localization mechanism.

### 4.1. Visual Macroprogramming

In addition to textual macroprogramming shown in Figures 1 and 2, STOP provides a visual macroprogramming environment. It leverages Google Maps (maps.google.com) to show the locations of sensor nodes as icons, and allows application developers to graphically specify a space where they observe. Figure 4 shows a pentagonal space (an observation area) on an example WSN deployed at the Boston Harbor. Given a graphical space definition, the STOP visual macroprogramming environment generates a skeleton macro-program that describes a set of points (pairs of longitude and latitude) constructing the space. Listing 5 shows a macro-program generated from a graphical space definition in Figure 4.

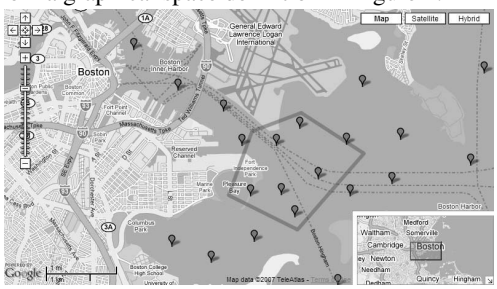



**Figure 4:** STOP Visual Macroprogramming Environment

**Listing 5:** A Generated Skeleton Code

```
1 points = [ # ( Latitude, Longitude )
2   Point.new( 42.35042512243457, -70.99880218505860 ),
3   Point.new( 42.34661907621049, -71.01253509521484 ),
4   Point.new( 42.33342299848599, -71.01905822753906 ),
5   Point.new( 42.32631627110434, -70.99983215332031 ),
6   Point.new( 42.34205151655285, -70.98129272460938 ) ]
7 s = Polygon.new( points )
```

### 4.2. STOP Runtime Environment

Once a macro-program is completed, it is transformed to a servlet (an application runnable on the STOP server) and interpreted by the JRuby interpreter in the STOP runtime environment (Figures 5 and 6). The STOP runtime environment operates the STOP server, STDB, gateway and Agent Repository. The STOP library is a collection of classes, closures (data/event handlers) and process objects (user-defined operators) that are used by STOP macro-programs. STDB stores node locations in `SensorLocations` table and the sensor data agents carry in `SensorData` table. Node locations are represented in the OpenGIS Well-Known Text (WKT) format[7].

When a STOP macro-program specifies a data query for the past, a SQL query is generated to obtain data from STDB.

[7] www.opengeospatial.org

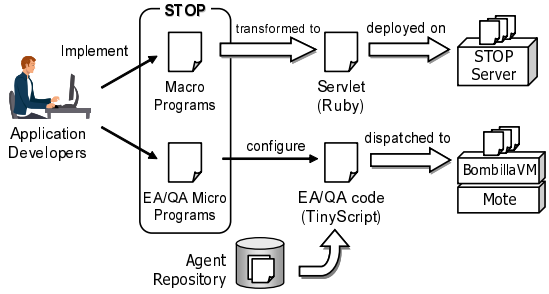



**Figure 5:** STOP Development Process

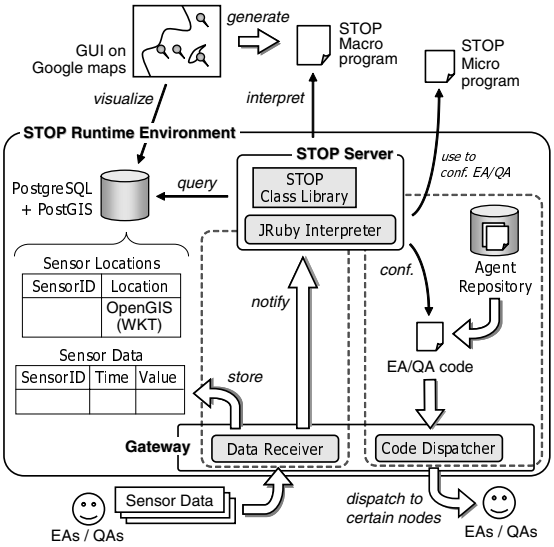



**Figure 6:** STOP Runtime Environment

`Spacetime::get_data()` implements this mapping from a data query to SQL query. Listing 6 shows an example SQL query. It queries ids, locations and sensor data from the nodes located in a certain space (`space` in Line 6). `Contains()` is an OpenGIS standard geographic function that examines if a geometry object (e.g., point, line and two dimensional surface) contains another geometry object. Also, this example query collects data from a given temporal domain (Lines 7 and 8). The result of this query is transformed to a Ruby object and passed to a corresponding data handler in a macro-program.

**Listing 6:** An Example SQL

```
1 SELECT SensorLocations.id, SensorLocations.location,
2        SensorData.value
3  FROM SensorLocations, SensorData
4  WHERE SensorLodations.id = SensorData.id AND
5        Contains(
6          space, SensorLocations.location ) = true AND
7        SensorData.time >= time - timeband AND
8        SensorData.time <= time + timeband;
```

If STDB does not have enough data that satisfy a data query's spatio-temporal resolution, QAs are dispatched to certain sensor nodes in order to collect extra sensor data. They carry the data back to STDB.

When a STOP macro-program specifies a future data query, QAs are dispatched to a set of nodes that meet the query's spatial resolution. `Spacetime::get_data()` imple-

ments this mapping from a data query to QA dispatch. After a QA is dispatched to a node, the QA periodically collects sensor data in a given temporal resolution. It replicates itself when it collects data, and the replicated QA carries the data to STDB. The data is passed to a corresponding data handler.

As shown above, the notion of spacetime allows application developers to seamlessly specify data collection for the past and future. Also, developers do not have to know whether STDB has enough data that satisfy the spatio-temporal resolutions that they specify.

### 4.3. In-Network Processing

As described in Section 3.3, STOP macroprogramming language supports user-defined data processing operators. `get_data()` can specify a data processing operator as its parameter (Section 3.1). Data processing is performed on the STOP server or in a network depending on data queries.

When a data query collects sensor data in the past and STDB can provide enough data, collected data is processed on the STOP server. Otherwise, a QA visits sensor nodes, collects sensor data, processes them on the last node of its route, and returns the result to a STOP macro-program. This in-network data processing saves the power consumption in a sensor network by reducing the amount of data to exchange between nodes. In STOP, to reduce the amount of data QAs brings, a QA is designed to have only its state and not to have code to execute on nodes. A code for in-network data processing is deployed only on the last node of QA's route, and a QA executes the code to process its data before returning to a base station. The STOP server transforms a code for in-network data processing in STOP macroprogramming language (Section 3.3) into TinyScript, and sends it to the last node of QA's route through the shortest path between a base station to the node before dispatching a QA.

### 4.4. Concurrency in the STOP Server

STOP macroprogramming language allows a STOP macro-program to have multiple data queries and data processing. This design strategy makes it easy to write queries and data processing which depend on results of precede data queries and data processing. However, without an appropriate threading model, i.e., if STOP macro-programs follow single thread model, they suffer from their low performance because data queries may take long time and block other data queries and data processing continually. To maximize the performance of STOP macro-programs, STOP macro-programs automatically create new threads so that multiple data queries and data processing perform in a parallel manner.

STOP macro-programs which deployed on the STOP server, i.e., servlets, can be invoked via SOAP, i.e., a XML-based protocol.As illustrated in Figure 7, a STOP macro-program (`Servlet`) starts when its `run` method is called. (`run` method is automatically generated during a transformation from a STOP macro-program to a servlet, and it is used to executes the original STOP macro-program.) Then, a new thread (`Data Collection Thread`) is created when a STOP macro-program calls `get_data()` so that it can perform a data collection in parallel with program's main thread. Each `get_data()` creates its own thread automatically. A data collection thread checks if STDB provides enough data, and collects data from STDB or dispatches QAs (Section 4.2). When a data collection thread dispatches QAs, it registers a corresponding event handler to a STOP macro-program. Once a gateway receives returning QA(s), it retrieves collected sensor data from the QA(s) and send it to the STOP server via SOAP (Figure 6). The STOP server notifies it to a STOP macro-program, and a STOP macro-program invokes the registered event handler.

Since a program's main thread and data collection threads run in parallel manner, `get_data()` may not be able to return a result to a program's main thread immediately. For example, in Listing 1, a variable `max_values` may not contain results of `get_data()` (Line 14) when a main thread accesses it (e.g., for drawing a graph or creating another data query based on the variable). In STOP, a main thread and a data collection thread are synchronized when a variable which contains a result of `get_data()` is accessed by a main thread.

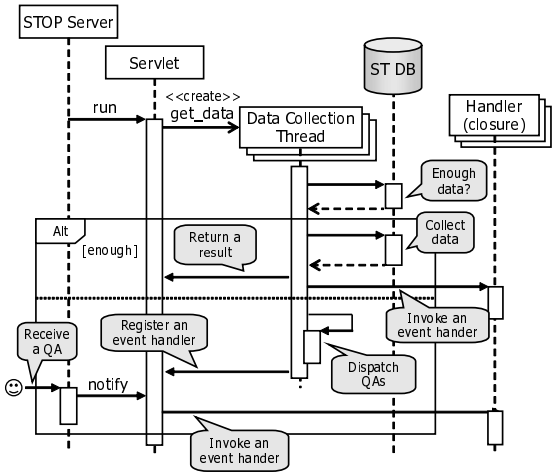



**Figure 7:** Concurrency in the STOP Server

## 5. STOP Microprogramming Language

STOP provides a *microprogramming* language to customize the default mapping between macro-programs and micro-programs. This allows developers to flexibly tune the performance and resource consumption of their applications by providing a means to tailor EAs and QAs. The STOP microprogramming and macroprogramming languages share the same syntax and semantics.

### 5.1. The Microprogramming for EA

By implementing `push_when` method on space, STOP microprogramming language allows specifying a condition when a EA replicates itself and a replicated agent starts migrating to a base station. Listing 7 shows an example microprogram for EA. Each node contained in `GLOBGALSPACE` periodically obtains its sensor data and executes `push_when` method. (The default period is one second.) `push_when` returns `true` or `false`. When it returns `true`, a EA replicates

itself on the node and a replicated EA starts migrating to a base station with sensor data. If it returns `false`, the node stores sensor data in its local storage (e.g., flush memory). A QA may visit to the node and collect the stored data in future. In Listing 7, each node periodically check whether local sensor data exceed 300nm or not.

**Listing 7:** A Micro-Program for EA (Local Filtering)

```
1 GLOBALSPACE.push_when{ |node|
2   node.get_data( 'f-spectrum' ) > 300 }
```

Listing 8 is another example of a micro-program for EA. In Listing 8, when a local sensor data exceeds 300nm, each node obtains a list of neighbors within one hop away (Line 4), and checks if the average of their sensor data exceed 300nm. This algorithm reduces the number of false positive sensor data compare with the algorithm in Listing 7 since each EA uses an average of neighbors' sensor data to decide whether to return a replicated EA, but may consume much energy since nodes exchange packets to obtains neighbors' sensor data.

**Listing 8:** A Micro-Program for EA (Neighthood Filtering)

```
1 GLOBALSPACE.push_when{ |node|
2   if node.get_data( 'f-spectrum' ) <= 300 then false
3
4   neighbors = node.get_neighbors_within(1)
5   total = node.get_data( 'f-spectrum' )
6   neighbors.each{ |neighbor|
7     total += neighbor.get_data( 'f-spectrum' ) }
8   total > 300 * (neighbors.size + 1); }
```

Listing 9 is another example. In Listing 9, if local sensor data exceed 300nm, each node broadcasts its local sensor data to one hop neighbors with a label `f-spectrum` (Line 4 and 5). Once receiving a broadcast message, each node keeps it as a tuple consisting of a source node id and a received value in a table of which name is `f-spectrum`. After that, a node retrieves sensor data from its `f-spectrum` table and checks if the average of them exceeds 300nm. Compare with the algorithm in Listing 8, this algorithm consumes less energy since it uses broadcasts to exchange sensor data instead of node-to-node communications.

**Listing 9:** A Micro-Program for EA (Gossip Filtering)

```
1 GLOBALSPACE.push_when{ |node|
2   if node.get_data( 'f-spectrum' ) <= 300 then false
3
4   node.bloadcast(
5     node.get_data( 'f-spectrum' ), 'f-spectrum', 1 )
6   total = node.get_data( 'f-spectrum' )
7   node.get_table( 'f-spectrum' ).each{ |node_id, value|
8     total += value }
9   total > 300 * (node.get_table.size + 1) }
```

As shown in Listing 7, 8 and 9, STOP microprogramming language allows defining arbitrary algorithm to decide when a replicated EA starts migrating to a base station. Depending on the requirements of WSN applications, e.g., low latency, low energy consumption or less false positive sensor data, application developers can implement their own algorithms and deploy them on nodes in a certain space.

### 5.2. Implementation of EA

STOP extends a Bombilla VM [9] and TinyScript to support mobile agents as one of messages which can move among sensor nodes with sensor data. A micro-program is used to configure EA code (template) in Agent Repository and a configured EA code is deployed on certain nodes by the STOP server (Figure 5 and 6). Listing 10 is an example EA code configured with Listing 7. A micro-program for EA is transformed into TinyScript and copied to a template. STOP server deploys this code on certain nodes in a space specified in a STOP macro-program.

**Listing 10:** A Fragment of EA Code in TinyScript

```
1 agent ea;
2 private data = get_sensor_data();
3 if (get_sensor_data() > 300) then
4   ea = create_event_agent();
5   set_source(ea, id());
6   set_sensor_data(ea, data);
7   return_to_basestation(ea);
8 end if
```

### 5.3. STOP Microprogramming for QA

In a default algorithm for QA's routing, only one QA visits to nodes in a certain space in the order of node's id. However, STOP microprogramming language allows implementing QA's routing algorithms such as Clarke-Wright Savings (CWS) algorithm [10].

CWS is a well known algorithm for Vehicle Routing Problem (VRP), one of NP-hard problems. The CWS algorithm is a heuristic algorithm which uses constructive methods to gradually create a feasible solution with modest computing cost. Basically, the CWS algorithm starts by assigning one agent per vertex (node) in the graph (sensor network). The algorithm then tries to combine two routes so that an agent will serve two vertices. The algorithm calculates the "savings" of every pair of routes, where the savings is the reduced total link cost of an agent after a pair of route is combined. The pair of routes that have the highest saving will then be combined if no constraint (e.g., deadline) is violated.

Listing 11 implements a QA's routing algorithm based on CWS. `CWS_ROUTING` is a process object which is executed right before dispatching QAs by the STOP server. The process object takes a set of nodes to visit (`nodes` in Line 2), a spatial resolution and a tolerable delay specified by a data query (`percentage` and `tolerable_delay`), and the maximum number of nodes an agent can visit (`max_nodes`). Since the size of the agent's payload is predefined, an agent is not allowed to visit and collect data from more than a certain number of nodes. The process object returns a set of sequences of nodes as routes on which each QA follows (`routes` in Line 9), e.g., if it returns three sequences of nodes, three QAs will be dispatched and each of them uses each sequence as its route.

In Listing 11, `CWS_ROUTING` selects part of nodes based on a spatial resolution (Line 9 to 12), and calculates savings of each adjacent nodes pair (Line 14 to Line 22). After that, routes are created by connecting two adjacent nodes in the order of savings. As described in Section 4, the STOP server stores the topology and latency information collected by EAs and QAs, and micro-programs can use that information through `node` object, e.g., `get_closest_node`, `get_shortest_path` and `get_delay` methods (Line 4 to 6).

**Listing 11:** An Micro-Program for QA (CWS Routing)

```
CWS_ROUTING = Proc.new{
 | nodes, percentage, tolerable_delay, max_nodes |

 closest = get_closest_node( base, nodes )
 delay = tolerable_delay/2 -
  closest.get_shortest_path(base).get_delay(base)

 # select closest nodes
 nodes = nodes.sort{|a, b|
   a.get_shortest_path(closest).get_delay <=>
   b.get_shortest_path(closest).get_delay}
   [0, (nodes.length * percentage/100).round - 1]

 nodes.each{ |node1|  # get savings of each pair
  nodes.each{ |node2|
    next if node1.get_hops(node2) != 1
    saving =
      node1.get_shortest_path(closest).get_delay +
      node2.get_shortest_path(closest).get_delay -
      node1.get_shortest_path(node2).get_delay
    savings[saving].push({node1, node2}) } }

 # connect nodes in the order of savings
 savings.keys.sort{ |saving|
  savings[saving].each{ |pair|
   if !pair[0].in_route && !pair[1].in_route ||
      pair[0].is_end != pair[1].is_end then
    route1 = pair[0].get_route_from(closest)
    route2 = pair[1].get_route_from(closest)
    if route1.get_delay <= delay &&
       route1.get_size <= max_nodes &&
       route2.get_delay <= delay &&
       route2.get_size <= max_nodes then
      pair[0].connect_with(pair[1]) # connect
    end
   end } }

 # return routes
 nodes.select{|node| node.is_end}
      .map{|node| node.get_route_from(closest)} }
```

### 5.4. Implementation of QA

A micro-program for QA is executed on the STOP server to configure QAs' routes. Each QA is implemented in TinyScript. As illustrated in Figures 5 and 6, QA's template code is stored in Agent Repository. The STOP server configures QA's route with a micro-program.

Listing 12 is a fragment of a configured QA code in TinyScript which is executed once at a base station. `set_agent_path` sets a path, i.e., a sequence of nodes to visit. `set_start_collecting` sets when to start collecting data by specifying an index of a node. In this example, a QA migrates from a base station to node 1 and 3, and collects data from nodes 11, 9 and 15. After visiting all nodes, the QA returns to a base station. A list of nodes to collect data is provided by a micro-program for QA. A list of nodes before starting a data collection (node 1 and 3) is the shortest path from a base station to the node to start collecting sensor data (nodes 11).

Also, `set_timestamp_from` and `set_timestamp_untill` specifies a time window of data to collect. STOP assumes timers of all nodes are synchronized and the STOP server can convert a representation of a time instant in a macro-program, i.e., absolute and relative times, into a clock of node.

**Listing 12:** A Fragment of QA Code

```
agent qa;
buffer path;
qa = create_query_agent();
path[]=1; path[]=3; path[]=11; path[]=9; path[]=15;
set_agent_path(qa, path);
set_start_collecting(qa, 2);
set_timestamp_from(qa, 100);
set_tiemstamp_untill(qa, 500);
migrate(qa);
```

Listing 13 is a fragment of a code deployed on each node beforehand, and used to accept QAs. It is executed when a node receives a broadcast message. (QAs are transmitted via broadcast.) It checks whether a QA collects data from the current node (Line 6). If the current node is the last one to visit, a QA executes a code for in-network processing (Section 4.3) and returns to a base station along the shortest path (Line 11 and 12). If not, a QA migrates to the next node (Line 14).

**Listing 13:** A Fragment of Code to Accept EAs

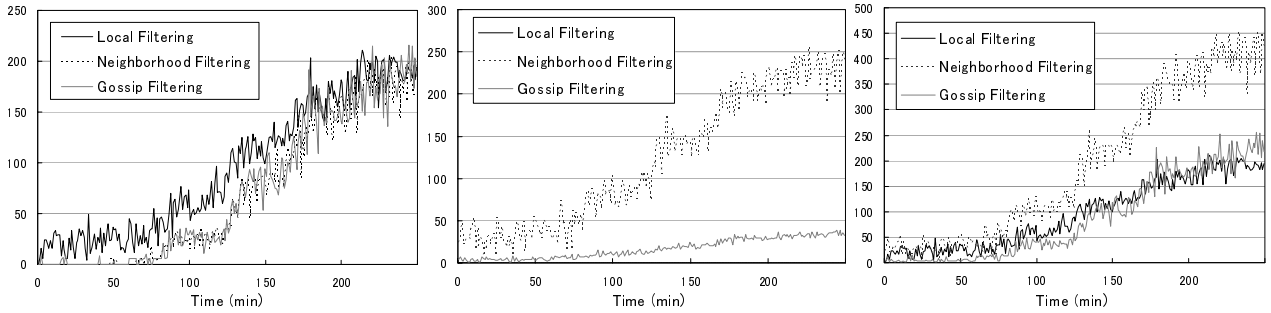
```
agent qa;
private node_id;
qa = migratebuf();  # retrieves a QA from a buffer
node_id = id();     # get the current node id

if (do_collection(qa, node_id)) then  # collect data?
  add_data(qa, get_sensor_data());
end if

if (is_end(qa, node_id)) then # the last node to visit?
  # do in-network processing here
  return_to_basestation(qa);
else
  migrate(qa);  # move to the next node
end if
```

## 6. Simulation Evaluation

This simulation study simulates a WSN deployed on the sea to detect oil spills in the Boston Harbor of Massachusetts. The WSN consists of nodes equipped with fluorometers. Nodes are deployed in an 6x7 grid topology in an area of approximately 620x720 square meters. They use MICA2 motes with the outdoor transmission range (radius) of 150 meters, 38.4kbps bandwidth, 128kB program memory (flush memory) and 2000 mAh battery capacity (two AA battery cells). The node running one of four WSN corners works as the base station. This study assumes that 100 barrels (approximately 3,100 gallons) of crude oil is spilled at the center of WSN. Simulation data set is generated with an oil spill trajectory model implemented in the General NOAA Oil Modeling Environment [11]. Sensor data shows a fluorescence spectrum of 280nm when there is no spilled oil, and it reaches 318nm when there exists oil. Each sensor has a white noise that is simulated as a normal random variable with its mean of zero and standard deviation of five percent of sensor data.

### 6.1. Event Detection

This section describes the performance differences between EA's algorithms shown in Listings 7 (Local Filtering), 8 (Neighborhood Filtering) and 9 (Gossip Filtering). Figure 8(a) and Figure 9 show the number of packets to transmit EAs to the base station and the number of false positive data. With Local Filtering, nodes decide whether to send replicated EAs independently; the base station receives many false positive data. With Neighborhood Filtering and Gossip Filtering, the base station receives mush less false positive data because nodes interact with each other before sending EAs. However, as shown in Figure 8(b), this interaction requires control overhead (i.e., power consumption). (There is no control overhead in Local Filtering.) Figure 8(c) shows the total number of packet transmissions. By reducing the number of




(a) Packet Transmission (b) Control Overhead (c) Total Packet Transmission

**Figure 8:** Packet Transmission

false positive data, the total number of packet transmissions is comparable in Gossip Filtering and Local Filtering.

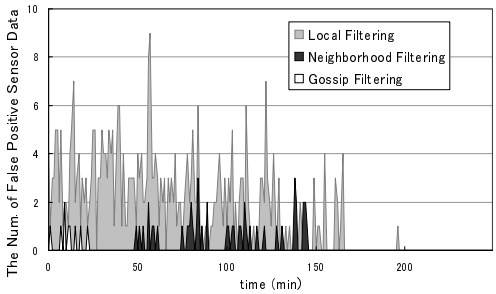



**Figure 9:** The Number of False Positive Sensor Data

## 6.2. Event Collection

As described in Section 4.2, QAs are dispatched to nodes to collect senor data when a query retrieves historical data and STDB cannot provide enough data.

Table 3 compares the behavior of different routing algorithms for QA, i.e., a default QA's routing algorithm and the CWS algorithm in Listing 11, when a query retrieves data from nodes in 3x3 nodes in the center of the WSN with 100% spatial-resolution and three minutes tolerable delay. The default QA's routing algorithm dispatches only one QA, and the QA simply visits to all nodes in the order of node's id. Since it does not consider query's timeliness, the result violates the tolerable delay specified by a query (i.e., three minutes). The CWS-based routing algorithm considers the tolerable delay and dispatches three QAs simultaneously, and one of QAs takes 2887ms to collect data and return to the base station (Other two take 2679ms and 2380ms). Since these three QAs' routes are partially over wrapped, especially a route between a base station and the area in where the 3x3 nodes are located, the total number of hops QAs take is larger than one of the default routing algorithm.

Depending on the requirements of WSN applications, e.g., timeliness and low energy consumption, application developers can implement their own algorithms for routing QAs by leveraging STOP microprogramming language.

In addition to queries for the past, QAs are used for queries for the future. Figure 10 shows the number of agents (sensor data carried by EAs and QAs) from 3x3 nodes in the center

**Table 3:** A Measurement on QA for the Past

| | **Default Routing** | **CWS Routing** |
|---|---|---|
| # of QAs | 1 | 3 |
| Latency (ms) | 4459 | 2887 |
| Total # of Hops | 26 | 48 |

of the WSN to the base station when a future query is used as in Listing 2. The temporal resolution of the future query is three minutes, i.e., obtain data every three minutes, and the spatial resolution varies from 0% to 100%. In addition to QAs, Gossip Filtering-based EAs are deployed on each node.

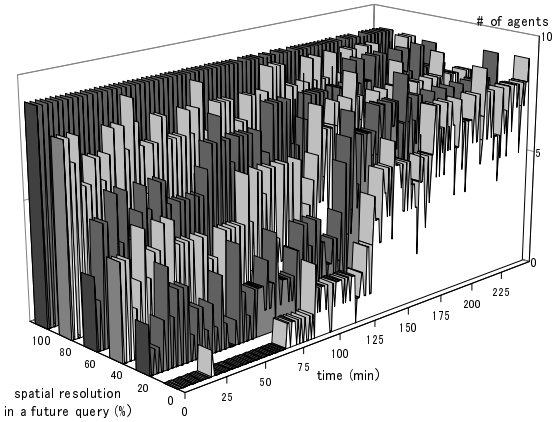



**Figure 10:** The Number of Sensor Data Received by the Base Station with a Future Query

When a spatial resolution is 0%, no future query is used, and few sensor data are transmitted during the first 75 minutes since only EAs send sensor data (replicated EAs). When a spatial-resolution is larger than 0%, deployed QAs send sensor data (replicated QAs) to the base station every three minutes according to a spatial-resolution even if deployed EAs do not send sensor data. In Figure 10, spikes appear every three minutes, and they correspond transmitted QAs. This way, a future query in STOP allows collecting sensor data to satisfy specified spatio-temporal resolutions even if there is no event.

## 6.3. Memory Footprint

Table 4 shows memory footprints on each sensor node. Total program sizes include TinyOS, Bombilla VM, and EA and QA code deployed on each node. As shown in Table 4, STOP is lightweight enough to run on MICA2 mote, which

has 128kB program memory. It runs on even much smaller sensor nodes, e.g. TelosB, which has only 48kB ROM.

**Table 4:** Memory Footprint

| EA Algorithms | Total (kB) | EA Code (kB) |
|---|---|---|
| Local Filtering | 41.3 | 0.077 |
| Neighborhood Filtering | 41.3 | 0.114 |
| Gossip Filtering | 41.3 | 0.116 |

## 7. Related Work

This work is an extension to the authors' previous work [12]. In this work, STOP is extended to support user-defined operators and micro-program customization, which were beyond the scope of the previous work. Moreover, this work evaluate and characterize the performance and resource consumption of applications built with STOP.

Kairos [13] and SNLong [14] provide programming abstractions to describe spatial relationships and data aggregation operations across nodes. Event detection, i.e., sending data to a base station, can be expressed without specifying the details of node-to-node communication and data aggregation. However, these languages require application developers to explicitly write programs to individual nodes. In contrast, STOP allows developers to program event detection to space-time as a global behavior of a WSN application. Also, Kairos and SNLong do not consider a temporal aspect of sensor data, i.e., they do not support data collection; data is always handled only at the current time frame.

Regiment [15] is another WSN macroprogramming language supporting in-network data processing and spatial-temporal event detection. It allows developers to specify arbitrary event detection algorithms, but Regiment does not support data collection and the notion of spatial and temporal resolutions. STOP supports data collection for both the future and past in arbitrary spatio-temporal resolutions.

TinyDB [16] extends SQL to support in-network data processing as well as spatio-temporal data collection. It allows application developers to program data collection for the future, but not for the past. Moreover, since TinyDB is an extension to SQL, its expressiveness is too limited to specify event handlers although it is well applicable to specify data queries. Therefore, developers need to learn and use an extra language to implement event handlers. In contrast, STOP supports spatio-temporal data collection for both the future and past. Its expressiveness is high enough to provide an integrated programming abstraction for data queries and event handlers. Also, TinyDB supports reactive data collections that are executed upon predefined events; however, it does not support event detection on individual nodes.

## 8. Conclusion

This paper proposes a new macroprogramming paradigm, called SpaceTime Oriented Programming (STOP). It is designed to reduce the complexity of WSN programming to specify spatio-temporal event detection and data collection. This paper describes how STOP macroprogramming and microprogramming are implemented and how the STOP runtime environment is implemented. This paper also characterizes the performance of applications built with STOP.

Several extensions are planned as future work. For example, an additional algorithm will be implemented to route QAs efficiently. It will consider combining QAs when they return the base station. This can reduce the number of packet transmissions, thereby power consumption.

## References


[1] U. C. Guard. Polluting Incident Compendium: Cumulative Data and Graphics for Oil Spills 1973-2004. Technical report, September 2006.

[2] L. Siegel. Navy Oil Spills. Webpage, November 1998.

[3] J. M. Andrews and S. H. Lieberman. Multispectral Fluorometric Sensor for Real Time in-situ Detection of Marine Petroleum Spills. In *The Oil and Hydrocarbon Spills, Modeling, Analysis and Control Conference*, July 1998.

[4] C. Brown, M. Fingas, and J. An. Laser Fluorosensors: A Survey of Applications and Developments. In *The Arctic and Marine Oil Spill Program Technical Seminar*, June 2001.

[5] M. Fingas and C. Brown. Review of Oil Spill Remote Sensors. *Spill Science & Technology Bulletin Journal*, 4(4), 1997.

[6] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, second edition, 1993.

[7] E. Meijer and P. Drayton. Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. *ACM OOPSLA Workshop on Revival of Dynamic Languages*, October 2004.

[8] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Comp. Surveys*, 37(4), 2005.

[9] P. Levis and D. Culler. Mate: A Tiny Virtual Machine for Sensor Networks. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[10] G. Clarke and J. W. Wright. Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, 4(12), 1964.

[11] C. Beegle-Krause. General NOAA Oil Modeling Environment (GNOME): A New Spill Trajectory Model. *Internation Oil Spill Conference*, March 2001.

[12] H. Wada, P. Boonma, and J. Suzuki. A SpaceTime Oriented Macro Programming Paradigm for Push-Pull Hybrid Sensor Networking. *IEEE ICCCN Workshop on Advanced Networking and Communications*, August 2007.

[13] R. Gummadi, O. Gnawali, and R. Govindan. Macroprogramming Wireless Sensor Networks using Kairos. *IEEE Int'l Conf. on Distributed Computing in Sensor Systems*, 2005.

[14] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein. Entirely Declarative Sensor Network Systems. *Int'l Conference on Very Large Data Bases*, September 2006.

[15] R. Newton, G. Morrisett, and M. Welsh. The Regiment Macroprogramming System. *Int'l Conference on Information Processing in Sensor Networks*, April 2007.

[16] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An Acqusitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 2005.