A SpaceTime Oriented Macroprogramming Paradigm for Push-Pull Hybrid Sensor Networking

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 a spatio-temporal macroprogramming paradigm for push-pull hybrid wireless sensor networks (WSNs). The proposed paradigm, called SpaceTime Oriented Programming (STOP), is designed to reduce the complexity of WSN programming by specifying data collection and processing 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 data collection and processing to spacetime, and abstracts away the details in WSNs, such as how many nodes are deployed in a WSN, how nodes are connected with each other and how to route data queries in a WSN. Using the notion of spacetime, data collection/processing is consistently specified for both the past and future in arbitrary spatio-temporal resolutions. This paper describes the design of the STOP language and the implementation of the STOP runtime environment.

1. Introduction

Macroprogramming is a programming paradigm that allows application developers to implement a WSN[1] application from a global viewpoint as a whole rather than a viewpoint of sensor nodes as individuals [1]. In general, a macroprogram specifies a global behavior of a WSN application. The macroprogram is transformed to node-level (micro) programs, and the micro programs are deployed on individual nodes. This way, macroprogramming is intended to increase the simplicity, productivity and reliability in WSN application programming.

This paper proposes a new macroprogramming paradigm, called SpaceTime Oriented Programming

(STOP). It provides a new programming abstraction to specify spatio-temporal data collection/processing as a global behavior of a WSN application rather than individual nodes' behaviors. STOP considers both spatial and temporal aspects of sensor data, and treats space and time as first-class citizens in macroprogramming. 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 application developers to program data collection and processing *to spacetime*, and abstracts away the low-level details in WSNs, such as how many nodes are deployed in a WSN, how nodes are connected and synchronized with each other, and how packets are routed among nodes. Using the notion of spacetime, data collection/processing are consistently specified for both the past and future in arbitrary spatio-temporal resolutions.

STOP assumes an application architecture that leverages mobile agents to collect and process sensor data in a push and pull hybrid manner (Figure 1). In this architecture, each WSN application is designed as a collection of agents, and there are two types of agents: *event agents* and *query agents*. An event agent (EA) is deployed on each node. When an EA detects an event (i.e., a significant change in its sensor reading), 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 the Agent Repository (Figure 1), and move to a certain spatial region (a certain set of nodes) to collect (or *pull*) sensor data that meet a certain temporal range. All collected sensor data through EAs and QAs are stored in a spatio-temporal database (STDB).

This paper is organized as follows. Section 2 presents a motivating WSN application that STOP is currently designed for. Section 3 describes how the STOP language is designed, and Section 4 describes how the STOP runtime environment is implemented. Sections 5 and 6 conclude
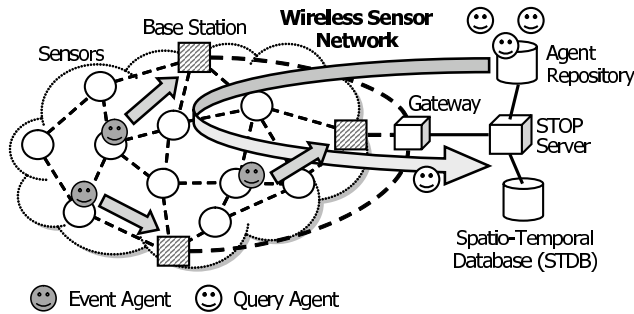
---

[1]Wireless Sensor Network (WSN)

Figure 1: A Sample WSN Organization

with some discussion on related work and future work.

2. A Motivating WSN Application

STOP operates on a multi-modal WSN[2], which consists of battery-operated sensor nodes and several base stations. STOP currently assumes that all nodes are stationary. Base stations are linked to the gateway that in turn connects to the STOP server (Figure 1). When EAs and QAs arrive at the STOP server, the STOP server extracts the sensor data the agents carry and store the data to STDB. Also, the STOP server maintains each node's location through a certain localization mechanism.

STOP is currently designed for coastal oil spill detection/monitoring. Oil spills occur frequently[3] 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 (e.g., fluorometers[4], surface roughness sensors[5], salinity sensors[6] and temperature sensors[7]) detects and monitors the spill. Oil may move and spread fast, 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 can provide real-time sensor data so that human operators effi-

ciently 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 less expensive than radar observation with aircrafts or satellites [7].

3. The STOP Language

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

Dynamic typing. Dynamic typing makes programs concise and readable by omitting type declarations and type casts [8]. This allows STOP developers to focus on their application logic without considering type-related housekeeping operations.

Object oriented programming. The notion of objects combines program states and functions, and modularizes their dependencies (i.e., which functions are supposed to change which states). It simplifies programs and improves their readability [9]. Moreover, since object oriented programming have been well accepted in many languages, it lowers the entry barriers and learning curve of the STOP language.

Extensibility for domain specific concepts. A programming language can substantially improve its expressiveness and ease of use by supporting domain specific concepts inherently in its syntax and semantics [10]. The STOP language requires an extensibility to support, as its primitives, various concepts specific to spatio-temporal data collection/processing (e.g., time, space, timespace, spatio-temporal resolutions, and data processing operators specific to sensor data).

Integration of a data/event and its handler. In STOP, application developers can write a data collection and a corresponding hander to process collected data. They can also write an event and a corresponding handler to respond to the event. The STOP language requires a mechanism to concisely express these pairs.

In order to satisfy the above requirements, the STOP language is designed as an extension to Ruby[8]. Ruby is an object oriented language supporting dynamic typing. It is also extensible to support domain specific concepts without changing its parsers and interpreters. The STOP language reuses Ruby's syntax and semantics, and introduces new

---

[2]A multi-modal WSN deploys multiple types of sensor nodes. Data from different types of nodes are aggregated, through in-network processing, to provide a multi-dimensional view of collected data.

[3]The US Coast Guard reports that 50 oil spills occurred in the US shores in 2004 [2], and the Associated Press reported that, on average, there was an oil spill caused by the US Navy every two days from fiscal year 1990 to 1997 [3]

[4]Fluorescence is a strong indication of the presence of oils. Certain compounds in oil absorb ultraviolet light, become electronically excited and fluoresce [4]. Different types of oil yield different fluorescent intensities (emission wavelengths) [5].

[5]Oil films locally damp sea surface roughness and give dark signatures, so-called slicks [4].

[6]Water salinity influences whether oil floats or sinks. Oil floats more readily in salt water. It also affects the effectiveness of dispersants [6].

[7]Water temperature impacts how fast oil spreads. Oil spreads faster in warmer water than in cold water [6].

language primitives specific to spatio-temporal data collection/processing. Also, Ruby supports *closures*, which modularize a code block as an object (similar to an anonymous method). The STOP language uses closures to define handlers and concisely associate them with data queries and events. Moreover, the STOP language assumes the JRuby interpreter[9] to execute STOP programs so that they can use existing Java libraries.

The STOP language supports *on-command* (one time or periodical) and *on-demand* (or event-driven) data collection. The two types of data collection are described in Sections 3.1 and 3.2.

3.1. On-Command Data Collection

On-command data collection is executed one time or periodically. It pairs a *data query* and a corresponding *data handler* to process obtained data. Listing 1 shows an example STOP program that specifies several on-command data collections. This program is visualized in Figure 2.

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 during the last one hour (`p`). STOP supports the concepts of absolute time and relative time, and allows application developers to denote relative time as a number annotated with a keyword such as `Week`, `Day`, `Hr`, `Min` and `Sec` (Line 4).

`get_space_at()` 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 data on at least 60% of sensor nodes (third parameter) in the space at 30 minutes before (the first parameter) with a 20 seconds time band (the second parameter).

`get_data()` is used to specify a data query. It is called on a space to query data available on the space (the first parameter) and process the data with a given operator (the second parameter). STOP currently supports several data aggregation operators 1. In Line 8, this method returns the average of fluorescence spectrum (`'f-spectrum'`) data from the space `s1`. The third parameter of `get_data()` specifies the tolerable delay (i.e., deadline) to collect and process data (the three minutes in this example).

`get_data()` can take a data handler as a closure. A code block from Line 9 to 11 is a closure that takes four parameters, `event_type`, `value`, `space` and `time`. In this example, these parameters contain a string `'f-spectrum'`, the average of fluorescence spectrum data from `s1`, the space `s1` and the time instant at 30 minutes before. Application developers write a data handler with these parameter values.

`get_spaces_every()` is called on a spacetime to return a discrete set of spaces that meet a certain spatio-temporal

Table 1: 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

resolution. In Line 13, this method returns spaces at every five minutes with the 10 seconds time band, and each space contains data on at least 80% of nodes it the space. Then, from Line 14 to 16, the maximum data is selected from each space. In STOP, a list has the `collect()` method[10], which takes a closure as its parameter, and the closure is called on each element in a list. In this example, each element in `spaces` is assigned to `space` parameter of a closure (Line 14). The maximum data on each space is selected by calling `get_data()`, and a set of results are contained in `max_values`. After that, `max_values` can be used for further processing. For example, drawing a graph by calling `draw_graph` which is implemented in Ruby or Java (Line 21). STOP supports `select()` method to return a subset of a list which meets a certain condition specified in a closure. From Line 24 to 28, `event_spaces` obtains subset spaces (from `spaces`), each of which yields 10 or lower standard deviation of sensor data and finds higher than 20 degrees difference in average data compared with a previous space in the list `spaces` (a space at five minutes before). From Line 30 to 34, this program focus on an individual node by calling `get_node()`. `get_data()` returns raw sensor data when its second parameter (data processing operator) is omitted.

Moreover, a closure allows enclosed code to access variables declared in the outside of the closure. It enables data queries/data processing to use results of precede data queries/data processing. In Line 17, a condition of an if statement refers the variable `avg_value` which contains a result of the precede query in Line 8. The variable `value` in Line 17 is bound to a closure and contains a result of a query in Line 15, i.e., the maximum value among collected fluorescence spectrum data on a space.

Listing 1: An Example STOP Program for an On-Demand 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 )
```

---

[10]In Ruby, a method that takes no parameter can be called without parentheses.

```
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     | event_type, value, space, time |
10    # the body of an event handler comes here.
11  }
12
13  spaces = sp.get_spaces_every( Min 5, Sec 10, 80 )
14  max_values = spaces.collect { |space|
15    space.get_data( 'f-spectrum', MAX, Min 2 ){
16      | event_type, value, space, time |
17      if value > avg_value then ...
18    }
19  }
20
21  name = 'f-spectrum'
22  event_spaces =
23    spaces.select{|s| s.get_data(name, STDEV, Min 5)<=10)}
24        .select{|s|
25          s.get_data(name,AVG,Min 5) -
26          spaces.prev_of(s).get_data(name, AVG, Min 5)>20)}
27
28  nodes_over_time =
29    event_spaces.collect{ |space| space.get_node(0) }
30  values = nodes_over_time.collect { |node|
31    node.get_data( name, Min 3 )
32  }
```
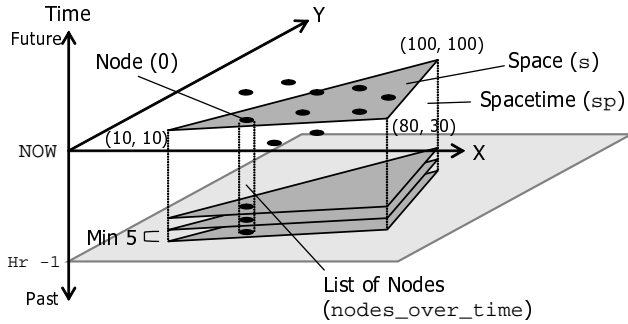

Figure 2: An Example On-Command Data Collection

3.2. On-Demand Data Collection

On-demand data collection is executed when an event (e.g., oil spill) occurs. It pairs an *event specification* and a corresponding *event handler* to respond to the event. Listing 2 shows an example STOP program that specifies an on-demand data collection. This program is visualized in Figure 3. Listing 2 is called when an event occurs. It investigates the event area for last 30 minutes in a low spatio-temporal resolution and monitors an area around the event area during the next one hour in a high spatio-temporal resolution.

This program is designed as an event handler of GLOBALSPACE. GLOBALSPACE is a special type of space, which represents a whole monitoring area, and it takes closures which implement an event handler and a condition to call the event handler. In Listing 2, when fluorescence spectrum data exceeds 300nm in some area (`select()` method in Line 2), it activates a closure of `on_event()` method (from Line 3 to 22). Parameters of a closure of `on_event()` represent features of an event (e.g., event area and time). Then, the

program (an event handler) creates a spacetime, `sp1`, consisting of `event_space` in which an event is found and a period during the last 30 minutes (Line 6). Then, a list of spaces at every six minutes (`past_spaces`) is extracted from `sp1` (Line 7), and from Line 8 to 10 counts the number of nodes that `'f-spectrum'` data exceeds 280nm on each space in `past_spaces`. This example also creates a spacetime in the future, `sp2`, consisting of the space `s2` during the next one hour (Figure 3). Line 18 collects the maximum `'f-spectrum'` data and executes a corresponding event handler (a closure) at every three minutes.

Listing 2: An Example STOP Program for an On-Demand Data Collection

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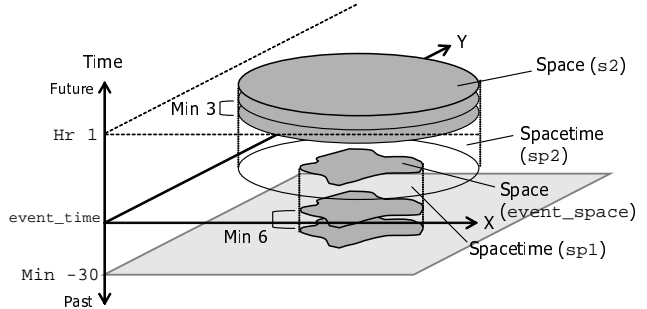

Figure 3: An Example On-Demand Data Collection

4. STOP Implementation

This section describes the details of a STOP implementation.

4.1. STOP Runtime Environment and GUI Tool

Implementation of STOP consists of a runtime environment and a supporting GUI tool as illustrated in Figure 4.
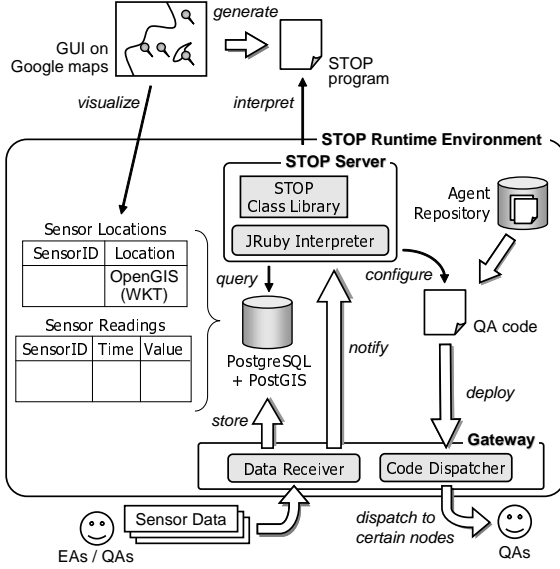
Figure 4: An Architecture of a STOP Implementation

STOP programs run on a STOP runtime environment, and a GUI tool helps developing STOP programs.

STOP provides a GUI leveraging google maps[11] to show sensors' locations as icons and allow application developers to specify a space where attracts the attention in a graphical manner (Figure 5). Since a space (e.g., a set of vertex) in a STOP program is identified in longitude and latitude, it is not easy to write a program without a support of a geographic information system. The STOP GUI generates a skeleton code of a STOP program which contains a set of coordinates constructing an area specified in the GUI. In Figure 5, a pentagonal shape is drawn at the mouth of the Charles River, Boston. It generates a skeleton code as in Listing 3. Application developers can start implementing their own code based on this skeleton code.

Figure 5: A Google Maps based GUI

Listing 3: A Skeleton Code

```
points = [
  # ( Latitude, Longitude )
  Point.new( 42.35042512243457, -70.99880218505860 ),
  Point.new( 42.34661907621049, -71.01253509521484 ),
  Point.new( 42.33342299848599, -71.01905822753906 ),
  Point.new( 42.32631627110434, -70.99983215332031 ),
  Point.new( 42.34205151655285, -70.98129272460938 ) ]
s = Polygon.new( points )
```

After application developers write a STOP program, the program is deployed on the STOP server and interpreted by a JRuby interpreter. STOP runtime environment has a STDB which stores pushed/pulled sensor data (in `SensorRedings` table) received through a gateway and locations of sensors (in `SensorLocations` table). When a STOP program has a query for the future, received sensor data is notified to an appropriate event handler in the program as well. Locations of sensors are represented in OpenGIS's Well-Known Text (WKT) format[12] in a STDB, and a STOP program uses geographic functions defined by OpenGIS to retrieve a list of sensor nodes in a certain space specified in a query. Listing 4 is an example SQL retrieving data from a STDB. This SQL query collects ids, locations and sensor data from sensors which located in a certain space (`space` in Line 6). `Contains()` is one of standard geographic functions defined by OpenGIS which examines if a geometry object (e.g., points, lines and two dimensional surfaces) contains another geometry object. Also, this query restricts time at when sensor data generated according to a STOP program (in Line 5 and 6). The result of this query is transformed into a Ruby object, and passed to a corresponding event handler in a STOP program.

Listing 4: An Example SQL

```
SELECT SensorLocations.id, SensorLocations.location,
       SensorReadings.value
  FROM SensorLocations, SensorReadings
 WHERE SensorLodations.id = SensorReadings.id AND
       Contains(
         space, SensorLocations.location ) = true AND
       SensorReadings.time >= time - timeband AND
       SensorReadings.time <= time + timeband;
```

If a STDB can provide enough sensor data to satisfy a query's spatio-temporal resolution, a program constructs a SQL query automatically and obtains data from a STDB. If a STDB can not provide enough sensor data, a STOP program automatically dispatch QAs to certain sensor nodes to "pull" sensor data in order to satisfy a spatio-temporal resolution through a gateway. STOP assumes each sensor node has a Bombilla VM [11] and QAs can migrate to sensor nodes to collect sensor data. Before dispatching QAs to a sensor network, a STOP program retrieves a QA code in TinyScript from an agent repository and configures it, e.g., specifying nodes to visit and data to collect (Section 4.3). Collected (pulled) data is also received through a gateway,

---

[11]maps.google.com/

[12]www.opengeospatial.org

and stored in a STDB. The notion of spatio-temporal resolution hides the details of a push-pull hybrid WSN architecture. Data collections are automatically performed according to required spatio-temporal resolutions. Application developers do not need to collect (pull) sensor data explicitly in their STOP programs.

`get_data()` can specify a data processing operator as its parameter (Section 3.1). Data processing is performed at a central server or in a network depending on data queries. When a data query collects sensor data in the past and a STDB can provide enough data, collected data is processed on a base station. Otherwise, QAs visit sensor nodes, collect and process sensor data in a network, and return the result to a STOP program which dispatched the QAs (Section 4.3). This in-network data processing reduces power consumption in a sensor network by reducing the amount of data to exchange between nodes. In either case, application developers do not need to know such details, i.e., how to collect data, and when and who to process data.

4.2. A Thread Model in STOP

STOP language allows a STOP program to have multiple data queries and data processing. This design strategy makes 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 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 programs, STOP programs automatically create new threads so that multiple data queries and data processing perform in a parallel manner.

Before running, STOP programs are transformed into servlets so that they can run on a STOP server. STOP programs which deployed on a STOP server can be invoked via SOAP[13]. As illustrated in Figure 6, a STOP program (`STOP Program`) starts when its `run` method is called. (`run` method is automatically generated during a transformation from a STOP program to a servlet, and the original STOP program is containded in the method.) Then, a new thread (`Data Collection Thread`) is created when a STOP 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 a STDB provides enough data, and collects data from a STDB or dispatches QAs (Section 4.1). When a data collection thread dispatches QAs, it registers a corresponding event handler to a STOP program. Once a gateway receives a returning QA, it retrieves collected sensor data from the QA and send it to a STOP server via SOAP (Figure 4). A

---

[13]A XML-based protocol [12]

STOP server notifies it to a STOP program, and a STOP 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 15) when a main thread calls `draw_graph()` (Line 21). 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. In Listing 1, a main thread automatically waits for completions of a data collection thread which returns `max_values` and calls `draw_graph()` (Line 21).
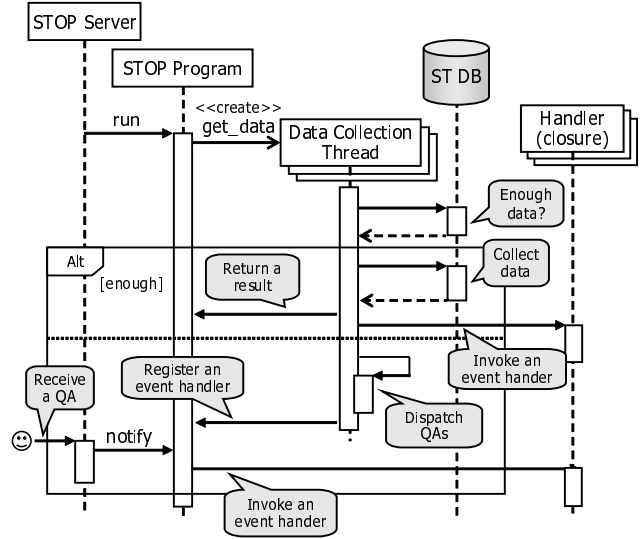

Figure 6: Threading Model in STOP

4.3. Design of Agents

As illustrated in Figure 4, a code of a QA is stored in an agent repository, and a STOP server configures and dispatches it to certain sensor nodes to collect sensor data. STOP extends a Bombilla VM and TinyScript to support mobile agents as one of messages which can move among sensor nodes with data, e.g., sensor readings. Listing 5 is a fragment of QA code.

Listing 5: A Fragment of QA Code

```
1  Once:
2  agent qa;
3  buffer path;
4  qa = create_query_agent();
5  path[]=1; path[]=3; path[]=11; path[]=9; path[]=15;
6  set_agent_path(qa, path);
7  set_start_collecting(qa, 2);
8  set_operator(qa, 1)  ! AVG
9  migrate(qa);
```

Listing 5 is deployed on only a base station and executed once. In this example, a QA collects data from

node of which IDs are 9, 11 and 15, i.e., nodes in a certain space, and calculates an average at each node. After visiting every node, a QA returns to 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 start collecting data from node 11 (`path[2] = 11`). `set_operator` sets an operator for in-network processing. (1 is a constant value representing SUM operator.) A STOP server sets a path (Line 5) and an operator (Line 8) according to a STOP program before deploying a code.

Listing 6 is a fragment of code deployed on each node. 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 9), and performs in-network processing according to an operator a QA specifies (Line 10 to 18). Then, if the current node is the last one to visit, a QA returns to a base station (Line 21 and 22). If not, a QA migrates to the next node (Line 24).

Listing 6: A Fragment of Code to Accept QAs

```
1   Broadcast:
2   agent qa;
3   private id;
4   private tmp;
5
6   qa = migratebuf();   ! retrieves a QA from a buffer
7   node_id = id();      ! get the current node id
8
9   if (do_collecting(qa, node_id)) then  ! collect data?
10   if (get_operator(qa) = 0) then  ! SUM
11    set_result(qa, get_result(qa) + get_sensor_data());
12   end if
13   if (get_operator(qa) = 1) then  ! AVG
14    tmp = get_result(qa) * (get_num_nodes_visited(qa)-1);
15    tmp = tmp + get_sensor_data();
16    set_result(qa, tmp / get_num_nodes_visited(qa) );
17   end if
18   ...
19  end if
20
21  if (is_end(qa, node_id)) then ! the last node to visit
22   return_to_basestation(qa);
23  else
24   migrate(qa);   ! move to the next node
25  end if
```

Unlike QAs, EAs are deployed on each node beforehand. As well as QAs, EAs are designed as messages and a code to decide whether to send a EA is written in TinyScript. Listing 7 is a fragment of EA code which sends a EA to a base station when a sensor reading exceeds 280. A EA code is executed periodically, and the period can be specified by calling `settimer()` function in TinyScript.

Listing 7: A Fragment of EA Code

```
1   agent ea;
2   private data = get_sensor_data();
3   if (get_sensor_data() > 280) 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. Related Work

This work is an extension to the authors' previous work [13]. In this work, STOP is extended to operate on a push-pull hybrid WSN architecture using EAs and QAs, while the previous work focused on operating STOP on a pull-based WSN architecture only with QAs. Moreover, this work newly investigates in-network data processing (Section 4.3), which was beyond the scope of the previous work. (The previous work focused on data collection that does not require in-network processing.) In addition, this work adds new facilities and mechanisms to the STOP server, such as a visual programming frontend and concurrency in the STOP server (Section 4.1).

Kairos [14] and SNLong [15] provide programming abstractions to describe spatial relationships and data aggregation operations across nodes. Data collection 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 data collection and processing to spacetime as a global behavior of a WSN application. Also, Kairos and SNLong cannot deal with a temporal aspect of sensor data; data is always handled only at the current time frame.

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, by leveraging closures, STOP allows developers to concisely associate a data query and a corresponding event handler. These language features increase the ease of programming and understanding the overall design of an WSN application.

Regiment [17] is another WSN macroprogramming language supporting in-network data processing and spatial-temporal data collection. It allows developers to specify data collection for the future, but not for the past. Also, Regiment does not support pull-based data collection and the notion of spatial and temporal resolutions. Unlike Regiment, STOP supports data collection for both the future and past in arbitrary spatio-temporal resolutions.

This work is the first attempt to investigate a push-pull hybrid WSN architecture that performs spatio-temporal data collection and processing. Most of existing push-pull hybrid WSNs do not address spatio-temporal aspects of sensor data, and they assume statically-assigned specific net-

work structures and topologies (e.g., star and grid topology) [18–20]. Therefore, data collection can be fragile against node/link failures. In contrast, STOP can operate in arbitrary network structures and topologies. It can implement failure-resilient queries by having the STOP server dynamically adjust the migration route that each QA follows. PRESTO [21] can perform push-pull hybrid data collection in arbitrary network structures and topologies. It also considers the temporal aspect in queries. However, it does not consider the spatial aspect in queries, and does not support queries for the future.

6. Conclusion

This paper proposes a new macroprogramming paradigm for push-pull hybrid WSNs, called SpaceTime Oriented Programming (STOP). Leveraging the notion of spacetime, STOP is designed to reduce the complexity of WSN programming to specify spatio-temporal data collection and processing. This paper describes how the STOP language is designed and how the STOP runtime environment is implemented.

Several extensions are planned as future work. In addition to pre-defined data processing operators such as AVG and MAX, STOP will be extended to allow application developers to define their own operators. This way, STOP can be more generic and customizable for various types of WSN applications.

The current GUI programming frontend supports the spatial aspect in data queries, but not the temporal aspect. It will be extended to allow developers to specify both the spatial and temporal aspects in data queries as shown in Figures 2 and 3. This will make it much easier to specify spatio-temporal queries so that even non-programmers (e.g., emergency responders and ocean scientists) can intuitively exploit WSNs without extensive learning process.

References

[1] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. *USENIX/ACM Symposium on Networked Systems Design and Implementation*, March 2004.

[2] US Coast Guard. Polluting Incident Compendium: Cumulative Data and Graphics for Oil Spills 1973-2004. Technical report, September 2006.

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

[4] 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.

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

[6] US Environmental Protection Agency. Understanding Oil Spills and Oil Spill Response. Technical report, 1999.

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

[8] 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.

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

[10] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, December 2005.

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

[12] World Wide Web Consortium. SOAP Version 1.2, June 2003.

[13] H. Wada, P. Boonma, and J. Suzuki. SpaceTime Oriented Macro Programming for Data Collection Sensor Networks. *Conference on Coastal Environmental Sensing Networks*, April 2007.

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

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

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

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

[18] W. Liu, Y. Zhang, W. Lou, and Y. Fang. Managing Wireless Sensor Networks with Supply Chain Strategy. *Int'l Conference on Quality of Service in Heterogeneous Wired/Wireless Networks*, October 2004.

[19] W. C. Lee, M. Wu, J. Xu, and X. Tang. Monitoring Top-k Query in Wireless Sensor Networks. *IEEE Int'l Conference on Data Engineering*, April 2006.

[20] S. Kapadia and B. Krishnamachari. Comparative Analysis of Push-Pull Query Strategies for Wireless Sensor Networks. *Int'l Conference on Distributed Computing in Sensor Systems*, June 2006.

[21] D. Ganesan M. Li and P. Shenoy. PRESTO: Feedback-Driven Data Management in Sensor Networks. *ACM/USENIX Symposium on Networked Systems Design and Implementation*, May 2006.