# Chronus: A Spatiotemporal Macroprogramming Language for Autonomic Wireless Sensor Networks

Hiroshi Wada[a], Pruet Boonma[b], Junichi Suzuki[c],

[a] National ICT Australia
Eveleigh, NSW 1430 Australia

[b] Department of Computer Engineering
Chiang Mai University
Chiang Mai, 50200, Thailand

[c] Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125-3393, USA

This chapter considers autonomic wireless sensor networks (WSNs) to detect and monitor spatiotemporally dynamic events, which dynamically scatter along spatiotemporal dimensions, such as oil spills, chemical/gas dispersions and toxic contaminant spreads. Each WSN application is expected to autonomously detect these events and collect sensor data from individual sensor nodes according to a given spatiotemporal resolution. For this type of autonomic WSNs, this chapter proposes a new programming paradigm, *spatiotemporal macroprogramming*, which is designed to reduce the complexity of programming event detection and data collection in autonomic WSNs. The proposed paradigm aids to specify them from a global network viewpoint as a whole rather than a viewpoint of sensor nodes as individuals and (2) make applications behave autonomously to satisfy given spatiotemporal resolutions for event detection and data collection. The proposed programming language, Chronus, treats space and time as first-class programming primitives and combines them as spacetime continuum. A spacetime is a three dimensional object that consists of two spatial dimensions and a time playing the role of the third dimension. Chronus allows application developers to program event detection and data collection to spacetime, and abstracts away low-level details in WSNs. The notion of spacetime provides an integrated abstraction for seamlessly expressing event detection and data collection as well as consistently specifying data collection for both the past and future in arbitrary spatiotemporal resolutions. This chapter describes Chronus' design, implementation, runtime environment and performance implications.

## 1. Introduction

Wireless sensor networks (WSNs) are considered as a key enabler to enhance the quality of monitoring and early warning in various domains such as environmental monitoring, emergency response and homeland security [1–3]. This chapter considers autonomic wireless sensor networks (WSNs) to detect and monitor spatiotemporally dynamic events, which dynamically scatter along spatiotemporal dimensions, such as oil spills, chemical/gas dispersions and toxic contaminant spreads. With autonomic WSNs, each application is expected to autonomously detect these events and collect sensor data from individual sensor nodes according to a given spatiotemporal resolution. Its goal is to provide human operators with useful information to guide their activities to detect and respond to spatiotemporally dynamic events. For example, upon detecting an event, a WSN application may increase spatial and temporal sensing resolutions to keep track of the event. In order to understand the nature of an event, another WSN application may collect past sensor data to seek any previous foretastes that have led to the current event.

This chapter proposes a new programming paradigm, called *spatiotemporal macroprogramming*, which is designed to aid spatiotemporal event detection and data collection with WSNs. This paradigm is designed to reduce the complexity of programming event detection and data collection by (1) specifying them from a global (or macro) network viewpoint as a whole rather than a micro viewpoint of sensor nodes as individuals and (2) making applications behave autonomously to satisfy given spatiotemporal resolutions for event detection and data collection. The proposed programming language, Chronus, treats space and time as first-class programming primitives. 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. Chronus 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 across nodes. The notion of spacetime provides an integrated abstraction to seamlessly express event detection and data collection for both the past and future in arbitrary spatiotemporal resolutions.

In Chronus, a macro-program specifies an application's global behavior. It is transformed or mapped to per-node micro-programs. Chronus is customizable to alter the default mapping between macro-programs and micro-programs and tailor micro-programs. It allows developers to flexibly tune the performance and resource consumption of their applications by customizing algorithmic details in event detection and data collection.

This paper is organized as follows. Section 2 overviews a motivating application that Chronus is currently implemented and evaluated for. Section 3 describes how Chronus is designed and how it is used to program spatiotemporal event detection and data collection. Section 4 presents how the Chronus runtime environment is implemented and how it interprets macro-programs to map them to micro-programs. Section 5 describes how Chronus allows for customizing the mapping from macro-programs to micro-programs. Section 6 shows simulation results to characterize the performance and resource consumption of applications built with Chronus. Sections 7 and 8 conclude with some discussion on related work and future work.

## 2. A Motivating Application: Oil Spill Detection and Monitoring

Chronus is designed generic enough to operate in a variety of dynamic spatiotemporal environments; however, it currently targets coastal oil spill detection and monitoring. Oil spills occur frequently[1] and have enormous impacts on maritime and on-land businesses, nearby res-

[1]The U.S. Coast Guard reports that 50 oil spills occurred in the U.S. shores in 2004 [4], and the Associated Press reported that, on average, there was an oil spill caused by the US Navy every two days from the fiscal year of 1990 to 1997 [5].

idents and the environment. Oil spills can occur due to, for example, broken equipment of a vessel and coastal oil station, illegal dumping or terrorism. Spilled oil may spread, change the direction of movement, and split into multiple chunks. Some chunks may burn, and others may evaporate and generate toxic fumes. Using an in-situ network of sensor nodes such as fluorometers[2], light scattering sensors[3], surface roughness sensors[4] salinity sensors[5] and water temperature sensors[6], Chronus aids developing WSN applications that detect and monitor oil spills. This chapter assumes that a WSN consists of battery-operated sensor nodes and several base stations. Each node is packaged in a sealed waterproof container, and attached to a fixed buoy.

In-situ WSNs are 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 away from spilled oil, alerting nearby facilities or evacuating people from nearby beaches [6, 7, 11]. In-situ WSNs can deliver more accurate information (sensor data) to operators than visual observation from the air or coast. Moreover, in-situ WSNs are more operational and less expensive than radar observation with aircrafts or satellites [11]. In-situ WSNs can operate during nighttime and poor weather, which degrade the quality of airborne and satellite observation.

## 3. Chronus Macroprogramming Language

Chronus addresses the following requirements for macroprogramming.

- **Conciseness.** The conciseness of programs increases the ease of writing and understanding. This can improve the productivity of application development. Chronus is required to facilitate concise programming.
- **Extensibility.** Extensibility allows application developers to introduce their own (i.e., user-defined) programming elements such as operators and functions in order to meet the needs of their applications. Chronus requires extensibility for developers to define their own operators used in a wide variety of applications.
- **Seamless integration of event detection and data collection.** Existing macroprogramming languages consider event detection and data collection largely in isolation. However, WSN applications often require both of them when they are designed to detect and monitor spatiotemporally dynamic events. Chronus is required to provide a single set of programming elements to seamlessly implement both event detection and data collection.
- **Complex event detection.** Traditional event detection applications often consider a single anomaly in sensor data as an event. However, in spatiotemporal detection and moni-

[2]Fluorescence is a strong indication of the presence of dissolved and/or emulsified polycyclic aromatic hydrocarbons of oils. Aromatic compounds absorb ultraviolet light, become electronically excited and fluoresce [6]. Different types of oil yield different fluorescent intensities [7].

[3]High intensity of light that is scattered by water indicates high concentration of emulsified oil droplets in the water [8, 9].

[4]Oil films locally damp sea surface roughness and give dark signatures, so-called slicks [6].

[5]Water salinity influences whether oil floats or sinks. Oil floats more readily in salt water. It also affects the effectiveness of dispersants [10].

[6]Water temperature impacts how fast oil spreads; faster in warmer water than cold water [10].

toring, it is important to consider more complex events, each of which consists of multiple anomalies. Chronus is required to express complex events in event detection.

- **Customizability in mapping macro-programs to micro-programs.** Different WSN applications have different requirements such as minimizing false positive sensor data, latency of event detection and power consumption. Therefore, the default mapping between macro-programs to micro-programs may not be suitable for a wide range of applications. Chronus is required to be able to customize the mapping according to application requirements.

Chronus is designed as an extension to Ruby[7]. 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 This simplifies programs and improves their readability [12]. Dynamic typing makes programs concise and readable by omitting type declarations and type casts [13, 14]. 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 [15]. To this end, Chronus 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 instead of building their own parsers or interpreters [16]. With this mechanism, Chronus reuses Ruby's syntax/semantics and introduces new keywords and primitives specific to spatiotemporal event detection and data collection such as time, space, spacetime and spatiotemporal resolutions.

Chronus leverages *closures*, which Ruby supports to modularize a code block as an object (similarly to an anonymous method). It uses a closure for defining an event detection and a corresponding handler to respond to the event as well as defining a data collection and a corresponding handler to process collected data. With closures, Chronus can concisely associate handlers with event detection and data collection specifications.

Chronus also employs *process objects*, which Ruby uses to encapsulate code blocks. It allows application developers to define their own (i.e., user-defined) operators as process objects.

Chronus simultaneously supports both data collection and event detection with WSNs. The notion of spacetime allows application developers to seamlessly specify data collection for the past and event detection in the future. It enables WSN applications to perform event detection and event collection at the same time.

Chronus allows developers to define three types of complex events for event detection: *sequence*, *any* and *all* events. Each complex event is defined with a set of events. A sequence event fires when a set of events occurs over time in a chronological order. An any and all event fire when one of or all of defined events occur(s), respectively.

Chronus leverages the notion of attribute-oriented programming to customize the default mapping from macro-programs to micro-programs. Attribute-oriented programming is a program marking technique to associate program elements with extra semantics [17, 18]. In Chronus, attributes are defined as special types of comments in macro-programs. Application developers can mark macro-program elements with attributes to indicate that the elements are

[7]www.ruby-lang.org

associated with certain application-specific requirements in, for example, packet rouging. The default macro-program-to-micro-program mapping can be customized by changing attributes that mark program elements in macro-programs.

## 3.1. Data Collection with Chronus

In Chronus, a data collection is executed one time or periodically to collect data from a WSN. A data collection pairs a *data query* and a corresponding *data handler* to process obtained data. Listing 1 shows an example macro-program that specifies several queries. Figure 1 visualizes this program.

A spacetime is created with the class `Spacetime` at Line 4. In Chronus, 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`). Chronus supports the concepts of absolute time and relative time. A relative time can be denoted as a number annotated with its unit (`Week`, `Day`, `Hr`, `Min` or `Sec`) (Line 3).

Listing 1: An Example Macro-Program for Data Collection

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

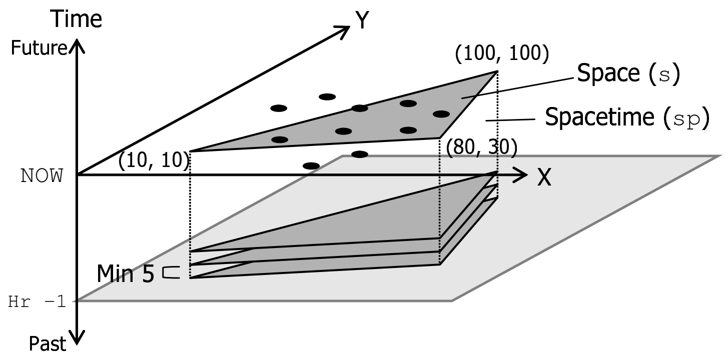


Figure 1. An Example Data Collection

`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 6, a snapshot 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 second time band (the second parameter).

Table 1
Key Methods in Chronus

| **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 a union/difference of two spacetimes |
| Space::get_data() | Executes a data query |
| Space::get_node() | Returns a particular node in a space |
| Space::get_nodes() | Returns a set of nodes in a space |
| Space::get_border() | Returns a set of nodes that are located at the border of a space |
| Space::divide() | Divides a space into sub spaces |
| Space +/- Space() | Returns a union/difference of two spaces |
| List::collect() | Returns a value from each element in a list |
| List::select() | Returns a subset of elements in a list |

Table 2
Data Aggregation Operators in Chronus

| **Operator** | **Description** |
|---|---|
| COUNT | Returns the number of collected data |
| MAX | Returns the maximum 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 |

`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). Chronus provides a set of data aggregation operators shown in Table 2. In Line 7, `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 8 to 9 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 given spatiotemporal resolution. In Line 11, this method returns spaces at every five minutes with the 10 second 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 12 and 13). In Chronus, a list has the `collect()` method[8], 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 14).

`select()` is used to obtain a subset of a list based on a certain condition specified in a closure. From Line 19 to 21, `event_spaces` obtains spaces, each of which yields 10 or lower standard deviation of data and finds an increase of 20 or more degrees in average in recent give minutes.

## 3.2. Event Detection with Chronus

An event detection application in Chronus pairs an *event specification* and a corresponding *event handler* to respond to the event. Listing 2 shows an example macro-program that specifies an event detection application. It is visualized in Figure 2. 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. The macro-program also collects sensor data from the event area over the next one hour in a high spatiotemporal resolution to monitor the oil spill.

Listing 2: An Example Macro-Program for an Event-based Data Query

```
 1  sp = Spacetime.new( GLOBALSPACE, Period.new( NOW, INF ) )
 2  spaces = sp.get_spaces_every( Min 10, Sec 30, 100 )
 3
 4  event spaces {
 5    sequence {
 6      not get_data('f-spectrum', MAX) > 290);
 7      get_data('f-spectrum', MAX) > 290;
 8      get_data('droplet-concentration', MAX) > 10;
 9      within MIN 30;
10    }
11    any {
12      get_data('f-spectrum', MAX) > 320;
13      window(get_data('d-concentration', MAX), AVG, HOUR -1) > 15;
14    }
15    all {
16      get_data('f-spectrum', AVG) > 300;
17      get_data('d-concentration', AVG) > 20;
18    }
19  }
20  execute{ |event_space, event_time|
21      # query for the past
22      sp1 = Spacetime.new(event_space, event_time, Min -30)
23      past_spaces = sp1.get_spaces_every(Min 6, Sec 20, 50)
24      num_of_nodes = past_spaces.get_nodes.select{ |node|
25          # @CWS_ROUTING
26          node.get_data('f-spectrum', Min 3) > 280}.size
27
28      # query for the future
29      s2 = Circle.new( event_area.centroid, event_area.radius * 2 )
30      sp2 = Spacetime.new( s2, event_time, Hr 1 )
31      future_spaces = sp2.get_spaces_every( Min 3, Sec 10, 80 )
32      future_spaces.get_data( 'f-spectrum', MAX, Min 1 ){
33        | data_type, value, space, time |
34        # data handler }
35    }
36  }
```

An event detection is performed against a set of spaces, which is obtained by `get_spaces_every()`. Line 1 obtains a spacetime `sp` that starts from the current time and covers whole observation area. `GLOBALSPACE` is a special type of space that represents the whole

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

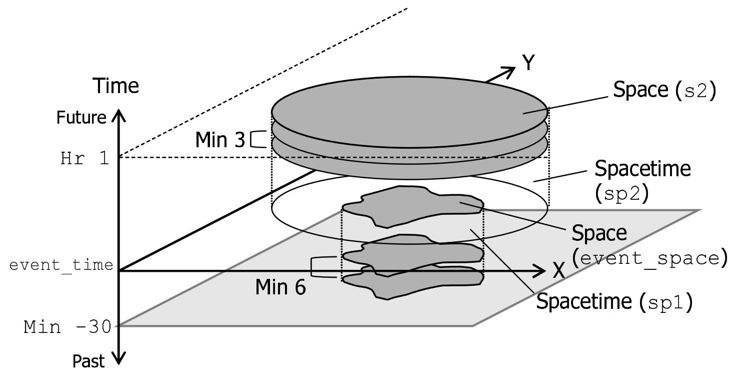


Figure 2. An Example Event-based Data Query

observation area, and `INF` represents the infinite future. `spaces` in Line 2 represents a set of `GLOBALSPACE`s at every 10 minutes with the 30 second time band in the future.

An event specification (Line 4 to 19) defines an event(s) (Line 5 to 18) and an event handler (Line 20 to 35). Listing 3 shows the language syntax to define event specifications. An event specification is declared with the keyword `event` followed by a set of spaces (Line 4). Events are defined as the conditions to execute a corresponding event handler. Chronus supports three event types: *sequence*, *any* and *all* events.

A *sequence* event fires when a set of *atomic event*s occurs in a chronological order. An atomic event is defined with `get_data()` or `window()` operation. `get_data()` returns spatially processed data; i.e., data aggregated over a certain space with one of operators in Table 2. In Listing 2, a sequence event is defined with three atomic events (Line 6 to 8). The sequence event fires if those atomic events occur chronologically in `spaces`. The first atomic event fires when the maximum fluorescence spectrum (`f-spectrum`) data does not exceed 290 (nm) (Line 6). The second atomic event fires when the maximum fluorescence spectrum (`f-spectrum`) data exceeds 290 (nm) (Line 7). The third one fires when the maximum concentration of oil droplet (`d-concentration`) exceeds 10 ($\mu L/L$) (Line 8). A sequential event can optionally specify a time constraint within which all atomic events occur. In Listing 2, three atomic events are specified to occur in 30 minutes (Line 9).

Listing 3: Event Specification Syntax

```
<query> = <event spec> <event handler>
<event spec> = 'event' <space> '{' {condition}+ '}'
<condition> = <sequence>  | <any> | <all>
<sequence> = 'sequence' '{' {<atomic event>}+ [<time restriction>] '}'
<atomic event> = (<get data> | <window>) <comparison operator> <number>
<time restriction> = 'within' <time>
<any> = 'any' '{' {<atomic event>}+ '}'
<all> = 'all' '{' {<atomic event>}+ '}'
<event handler> = 'execute' '{' <Chronus code> '}'
```

A sequence condition is transformed into a state machine in which each state transition corresponds to an atomic event. For example, a sequence in Listing 2 is transformed to a state machine in Figure 3. `A` is the initial state, and it transits to `B` when the first atomic event, `not get_data('f-spectrum', MAX) > 290`, occurs. The sequence event's handler is executed in `D`.

An *any* event fires when one of defined atomic events occurs. Listing 2 uses `window()` to define an atomic event. `window()` returns temporally processed data; i.e., data aggregated over time with one of operators in Table 2. In Listing 2, an any event is defined with two atomic

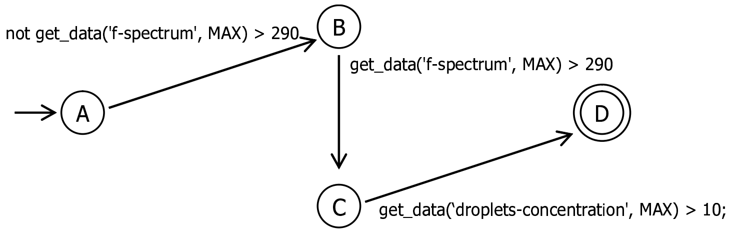

not get_data('f-spectrum', MAX) > 290
get_data('f-spectrum', MAX) > 290
get_data('droplets-concentration', MAX) > 10;
A
B
C
D

Figure 3. A State Transition transformed from Listing 2

events (Line 12 and 13). It fires and executes a corresponding event handler if one of the two atomic events occurs. The first atomic event fires when the maximum fluorescence spectrum (`f-spectrum`) data exceeds 320 (nm). The second atomic event fires when the average of the maximum oil droplet concentration (`d-concentration`) exceeds 15 ($\mu L/L$) in recent one hour.

An *all* event fires and executes a corresponding event hander when all of defined atomic events occur. It does not consider a chronological order of atomic events as a sequence event does. Listing 2 defines two atomic events (Line 16 and 17).

An event handler is specified as a closure of `execute` (Line 20 to 36). Its parameters are a space where an event has occurred and the time when the event has occurred (Line 20). In Line 23, a spacetime (`sp1`) is created to cover `event_space` over the past 30 minutes. Then, `sp1` is used to examine how many nodes have observed 280 (nm) or higher fluorescence spectrum every six minutes. Line 25 is a special comment that contains an *attribute*, which starts with @. It is used to customize the default mapping from a macro-program to a micro-program. The attribute `@CWS_ROUTING` declares to use an algorithm called CWS for packet routing. (See Section 5.3 for more details about CWS.) In Line 36, another spacetime (`sp2`) is created to specify a data collection in the future. `sp2` covers a larger space than `event_space` for an hour in the future. It is used to examine the maximum fluorescence spectrum every three minutes.

Listing 2 uses `get_spaces_every()` to obtain a set of spaces (Line 2). This operation guarantees that a Chronus application can watch a certain space with a certain spatiotemporal resolution. However, it does not suit for detecting highly critical events since an event may happen between two spaces in a list that `get_spaces_every()` returns and an event handler fails to respond to the event immediately. For example, when `get_spaces_every()` returns a set of spaces every 30 minutes, an application may respond to an event 30 minutes after the event happens. Reducing the interval of time to watch (i.e., the first parameter of `get_spaces_every()`), however, results to consume unnecessarily large amount of battery power since `get_spaces_every()` may send packets to sensor nodes in a WSN to collect data (see details in Section 4.2). In order to avoid this situation, Chronus supports `get_spaces_upon()` operation to obtain a set of spaces in addition to `get_spaces_every()`. Chronus assumes that each sensor node sends back data to a base station when a condition met (e.g., fluorescence spectrum exceeds 290(nm), and `get_spaces_upon()` creates a space upon the arrival of data from sensor nodes.

Listing 4: An Example program using `get_spaces_upon()`

```
sp = Spacetime.new( GLOBALSPACE, Period.new( NOW, INF ) )
spaces1 = sp.get_spaces_upon( Sec 30 )

sp = Spacetime.new( space, Period.new( NOW, INF ) )
spaces2 = sp.get_spaces_every( Min 10, Sec 30, 100 )

observing_space = spaces1 + spaces2
```

Listing 4 shows an example using `get_spaces_upon()` and `get_spaces_every()`. `spaces1` contains a set of spaces created upon the arrival of data from nodes (Line 2), and `spaces2` contains a set of spaces that satisfies a certain spatiotemporal resolution (Line 5). Chronus treats both types of spaces in the same way and allows for combining them (Line 7). With the combined spaces, `observing_space`, an application can monitor an area with a certain spatiotemporal resolution and respond to events that occur in the area immediately.

## 3.3. User-defined Data Aggregation Operators

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

Listing 5 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 (i.e., a set of sensor data to process) to each operator (Lines 1, 9 and 13).

Listing 6 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 5: Predefined Data Aggregation 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 6: An Example User-defined Operator for Data Aggregation

```
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. Chronus Implementation

Chronus is currently implemented with an application architecture that leverages mobile agents in a push and pull hybrid manner (Figure 4). 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 4), 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 Chronus server, it extracts the sensor data the agents carry, and stores the data to a spatiotemporal database (STDB).

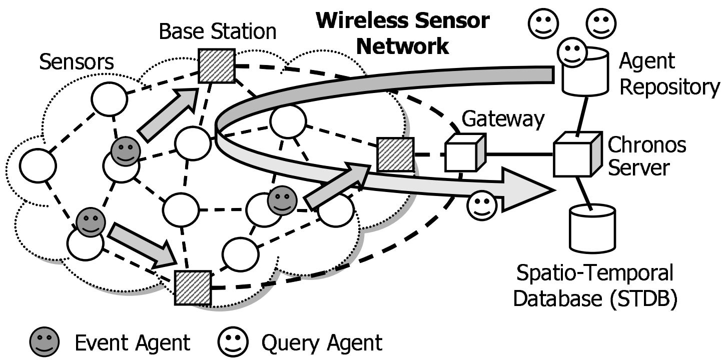


Figure 4. A Sample WSN Organization

At the beginning of a WSN operation, the Chronus 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 Chronus server, they update the topology and latency information that the Chronus server maintains. The Chronus 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, Chronus 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 5 shows a pentagonal space (an observation area) on an example WSN deployed at the Boston Harbor. Given a graphical space definition, the Chronus visual macroprogramming environment generates a skeleton macro-program that describes a set of points (pairs of longitude and latitude) constructing the space. Listing 7 shows a macro-program generated from a graphical space definition in Figure 5.

Listing 7: A Generated 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 )
```

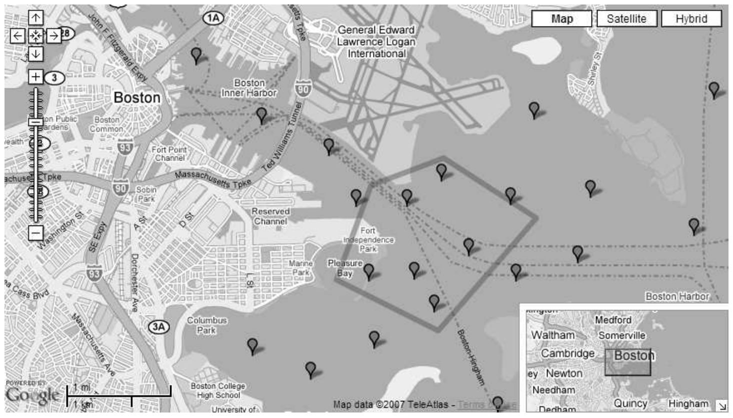


Figure 5. Chronus Visual Macroprogramming Environment

## 4.2. Chronus Runtime Environment

Once a macro-program is completed, it is transformed to a servlet (an application runnable on the Chronus server) and interpreted by the JRuby interpreter in the Chronus runtime environment (Figures 6 and 7). The Chronus runtime environment operates the Chronus server, STDB, gateway and Agent Repository. The Chronus library is a collection of classes, closures (data/event handlers) and process objects (user-defined operators) that are used by Chronus 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[9].

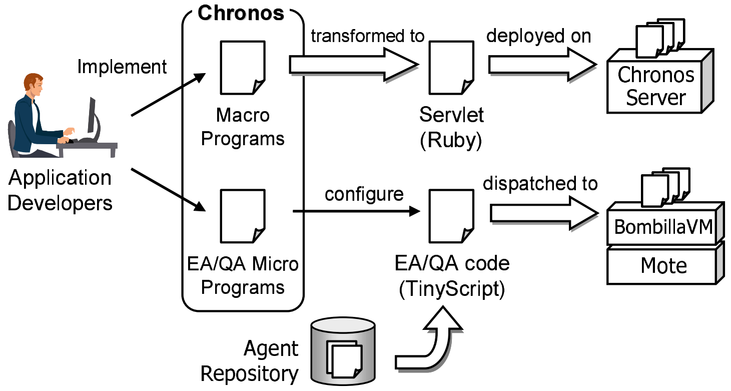


Figure 6. Chronus Development Process

When a Chronus macro-program specifies a data query for the past, a SQL query is generated to obtain data from STDB. `get_data()` implements this mapping from a data query to SQL query. Listing 8 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.

[9]www.opengeospatial.org

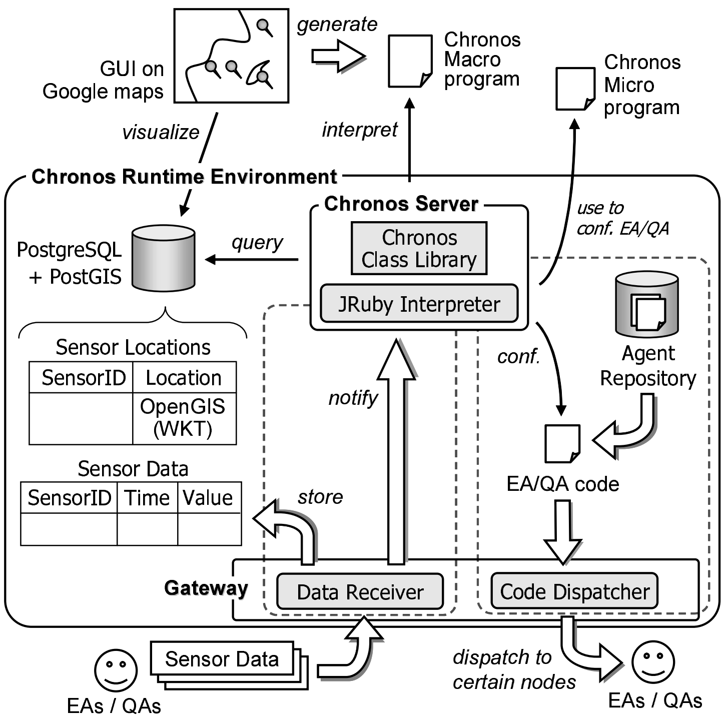


Figure 7. Chronus Runtime Environment

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

When a Chronus macro-program specifies a future data query, QAs are dispatched to a set of nodes that meet the query's spatial resolution. `get_data()` implements 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 spatiotemporal resolutions that they specify.

Listing 8: An Example SQL

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

## 4.3. In-Network Processing

As described in Section 3.3, Chronus 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 Chronus 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 Chronus 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 Chronus 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 Chronus, 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 Chronus server transforms a code for in-network data processing in Chronus macroprogramming language (Section 3.3) into TinyScript, and sends it to the last node of QA's route through the shortest path from a base station to the node before dispatching a QA.

## 4.4. Concurrency in the Chronus Server

Chronus macroprogramming language allows a Chronus 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 preceding data queries and data processing. However, without an appropriate threading model, i.e., if Chronus 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 Chronus macro-programs, Chronus macro-programs automatically create new threads so that multiple data queries and data processing perform in a parallel manner.

Chronus macro-programs which deployed on the Chronus server, i.e., servlets, can be invoked via SOAP, i.e., a XML-based protocol [19]. As illustrated in Figure 8, a Chronus macro-program (`Servlet`) starts when its `run()` method is called. (`run()` method is automatically generated during a transformation from a Chronus macro-program to a servlet, and it is used to execute the original Chronus macro-program.) Then, a new thread (`Data Collection Thread`) is created when a Chronus 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 Chronus macro-program. Once a gateway receives returning QA(s), it retrieves collected sensor data from the QA(s) and send it to the Chronus server via SOAP (Figure 7). The Chronus server notifies it to a Chronus macro-program, and a Chronus 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 Chronus, 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.

## 5. Chronus Microprogramming Language

Chronus provides a *microprogramming* language to customize the default behavior of micro-programs. While Chronus macro-programs are always written on spacetime, Chronus micro-programs are always written on nodes. This allows developers to flexibly tune the performance

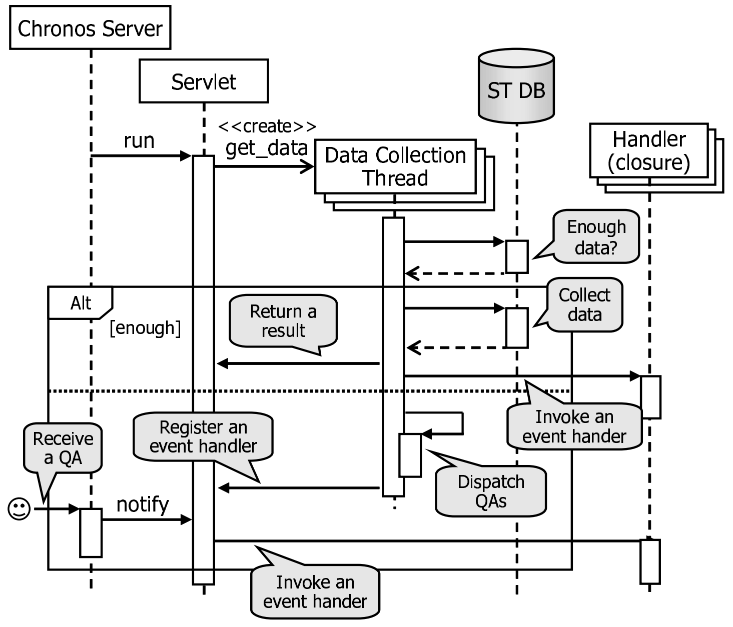


Figure 8. Concurrency in the Chronus Server

and resource consumption of their applications by providing a means to tailor EAs and QAs. The Chronus microprogramming and macroprogramming languages share the same syntax and semantics.

## 5.1. Microprogramming EAs

By implementing a process object, Chronus microprogramming language allows specifying a condition when an EA replicates itself and a replicated agent starts migrating to a base station. Listing 9 shows an example micro-program for EAs. Each node periodically obtains its sensor data and executes `LOCAL_FILTERING` process object. A process object returns `true` or `false`. When it returns `true`, an 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 9, each node periodically check whether local sensor data exceed 300(nm) or not.

Listing 9: A Micro-Program for EAs (Local Filtering)

```
1 LOCAL_FILTERING = proc do |node|
2   node.get_data( 'f-spectrum' ) > 300
3 end
```

By leveraging attribute-oriented programming, Chronus allows to use micro-programs in macro-programs while keep separating them. Listing 10 is an example Chronus macro-program that refers a micro-program for EAs through the use of an attribute. A keyword starts with `@` in a comment, e.g., `@LOCAL_FILTERING` in Line 1, is called an `attribute`, and the attribute `marks` the following program element. In Listing 10, each node contained in the spacetime (`sp`) uses `LOCAL_FILTERING` (Listing 9) to decide when to send back data to a base station. If no attribute is specified right before an initialization of a specetime, nodes in the spacetime do not send back data to a base station (only QAs collect data from nodes). Attribute-oriented programming improves the readability and maintainability of Chronus programs by separating macro-programs and micro-programs clearly while providing a mechanism to combine them.

Listing 10: A Macro-Program for Deploying A Micro-Program for EAs

```
# @LOCAL_FILTERING
sp = Spacetime.new( GLOBALSPACE, Period.new( Hr -1, Min 30 ) )
spaces = sp.get_spaces_every( Min 10, Sec 30, 100 )
```

Listing 11 is another example of a micro-program for EAs. In Listing 11, when a local sensor data exceeds 300(nm), each node obtains a list of neighbors within one hop away (Line 4), and checks if the average of their sensor data exceed 300(nm). This algorithm reduces the number of false positive sensor data compare with the algorithm in Listing 9 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 11: A Micro-Program for EAs (Neighborhood Filtering)

```
1  NEIGHBORHOOD_FILTERING = proc do |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);
9  end
```

Listing 12 is another example. In Listing 12, if local sensor data exceed 300(nm), 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 300(nm). Compare with the algorithm in Listing 11, this algorithm consumes less energy since it uses broadcasts to exchange sensor data instead of node-to-node communications.

Listing 12: A Micro-Program for EAs (Gossip Filtering)

```
 1  GOSSIP_FILTERING = proc do |node|
 2    if node.get_data( 'f-spectrum' ) <= 300 then false
 3
 4    node.broadcast(
 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)
10  end
```

As shown in Listing 9, 11 and 12, Chronus 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 EAs

Chronus extends a Bombilla VM [20] 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 Chronus server (Figure 6 and 7). Listing 13 is an example EA code configured with Listing 9. A micro-program for EAs is transformed into TinyScript and copied to a template. Chronus server deploys this code on certain nodes in a space specified in a Chronus macro-program.

Listing 13: A Fragment of EA Code in TinyScript

```
agent ea;
private data = get_sensor_data();
if (get_sensor_data() > 300) then
  ea = create_event_agent();
  set_source(ea, id());
  set_sensor_data(ea, data);
  return_to_basestation(ea);
end if
```

### 5.3. Microprogramming QAs

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, Chronus microprogramming language allows implementing QA's routing algorithms such as Clarke-Wright Savings (CWS) algorithm [21].

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 14 implements a QA's routing algorithm based on CWS. `CWS_ROUTING` is a process object which is executed right before dispatching QAs by the Chronus 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. Moreover, a process object returns a set of sequences of nodes, a QA replicates itself on an intermediate node and visit nodes in parallel. For example, when a process returns a set of two sequences of nodes as `{{5, 9, 10}, {5, 7}}`, a QA moves from a base station to node 5 and replicates itself. One QA visits to nodes 9 and 10, and the other visits to node 7. After that, the two QAs merge into one QA, and it returns to a base station.

In Listing 14, `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 Chronus 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 14: A Micro-Program for QAs (CWS Routing)

```
CWS_ROUTING = proc do
 | 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)}
end
```

As well as micro-programs for EA described in Section 5.1, micro-programs for QA can be referred in macro-programs through the use of attributes. Listing 15 is an example Chronus macro-program that refers a micro-program for QAs through the use of an attribute. A micro-program for QAs is used as a default in a spacetime when a corresponding attribute marks an initialization of the spacetime (Line 1 and 2). Also, a micro-program for QAs is used only for performing a certain `get_data()` when a corresponding attribute marks a `get_data()` method call (Line 1 and 2).

Listing 15: A Macro-Program for Deploying a Micro-Program for QAs

```
1  # @CWS_ROUTING
2  sp = Spacetime.new( GLOBALSPACE, Period.new( Hr -1, Min 30 ) )
3  spaces = sp.get_spaces_every( Min 10, Sec 30, 100 )
4
5  max_values = spaces.collect { |space|
6    # @CWS_ROUTING
7    space.get_data( 'f-spectrum', MAX, Min 2 ){
8      | data_type, value, space, time |
9      # data handler
10   }
11 }
```

## 5.4. Implementation of QAs

A micro-program for QAs is executed on the Chronus server to configure QAs' routes. Each QA is implemented in TinyScript. As illustrated in Figures 6 and 7, QA's template code is stored in Agent Repository. The Chronus server configures QA's route with a micro-program.

Listing 16 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 (Line 5 and 6). `set_start_collecting()` sets when to start collecting data by specifying an index of a node (Line 7). If multiple QAs are used to collect data in parallel, each QA's route is specified in the sequence delimited with 0. In this example, a QA migrates from a base station to node 1 and 3, and starts collecting data. At node 3, a QA creates another QA, and one QA collects data from nodes 11 and 9, and another QA collects data from nodes 10 and 13 (Line 5). After visiting all nodes, each QAs returns to the node they split, node 3 in this example, and merge themselves. In this example, two QAs (the first and the second QAs) are merged into one QA. (Line 9 and 10). A list of nodes to collect data is provided by a micro-program for QAs. 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 12). Also, `set_timestamp_from()` and `set_timestamp_untill()` specifies a time window of data to collect. Chronus assumes timers of all nodes are synchronized and the Chronus 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 16: A Fragment of QA Code

```
 1  agent qa;
 2  buffer path;
 3  buffer merge;
 4  qa = create_query_agent();
 5  path[]=1; path[]=3; path[]=11; path[]=9; path[]=0; path[]=10; path[]=13;
 6  set_agent_path(qa, path);
 7  set_start_collecting(qa, 1);
 8  set_agent_num_of_qa(qa, 2);
 9  merge[] = 1; merge[] = 2;
10  set_agents_merge(qa, merge);
11  set_timestamp_from(qa, 100);
12  set_tiemstamp_untill(qa, 500);
13  migrate(qa);
```

Listing 17 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 17: A Fragment of Code to Accept EAs

```
 1  agent qa, copy;
 2  buffer path;
 3  private node_id;
 4  qa = migratebuf();  # retrieves a QA from a buffer
 5  node_id = id();     # get the current node id
 6
 7  # start collecting data?
 8  if start_collection(qa, node_id) then
 9    for i = 1 to get_num_of_qa(qa) - 1
10      copy = create_query_agent();
11      path = get_path(copy, i);
```

```
    set_agent_path(copy, path);
    set_start_collecting(copy, 1);
    migrate(copy);
  next i
end if

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. (Figure 9). 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 4,200 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 [22]. Sensor data shows a fluorescence spectrum of 280(nm) when there is no spilled oil, and it reaches 318(nm) 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.

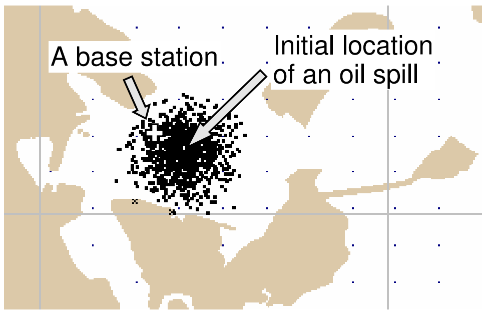


Figure 9. A Simulated Oil Spill

### 6.1. Event Detection

This section describes the performance differences between EA's algorithms shown in Listings 9 (Local Filtering), 11 (Neighborhood Filtering) and 12 (Gossip Filtering). Figure 10 and Figure 13 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 11, this interaction requires control overhead (i.e., power consumption). (There is no control overhead in Local Filtering.) Figure 12 shows the total number of packet transmissions. By reducing the number of false positive data, the total number of packet transmissions is comparable in Gossip Filtering and Local Filtering.

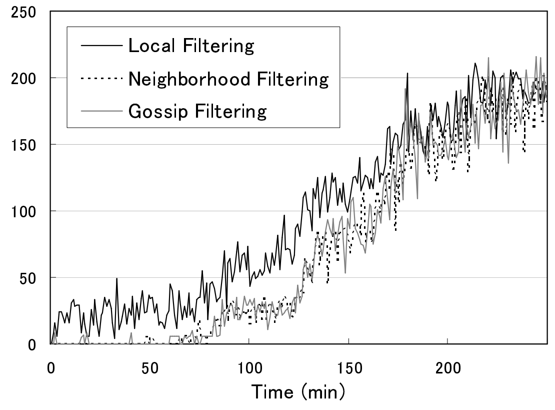


Figure 10. Packet Transmission

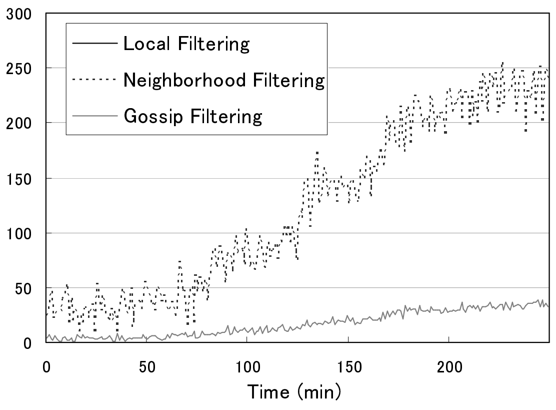


Figure 11. Control Overhead

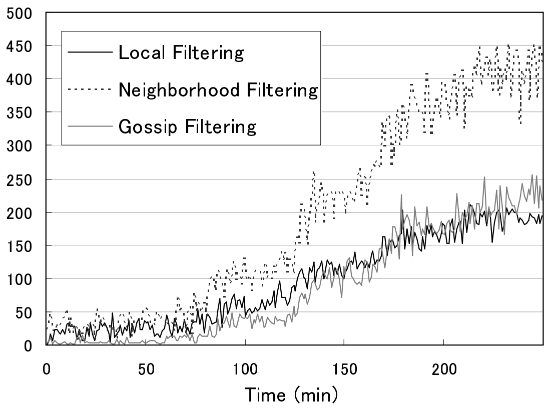


Figure 12. Total Packet Transmission

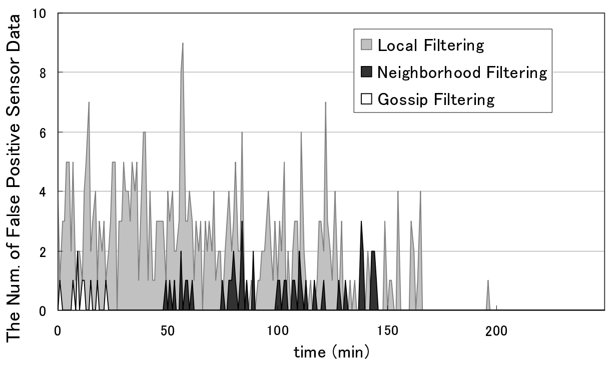


Figure 13. The Number of False Positive Data

## 6.2. Data Collection in the Future

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 14, 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. Each QA can contain 13 sensor readings. 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 in Section 5.3 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 (battery consumption) is larger than one of the default routing algorithm. When an extended version of CWS-based routing algorithm is used, the total number of hops drops significantly since a QA replicates itself and replicated QAs merge on an intermediate node. The total number of hops QAs take is almost same as 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 Chronus microprogramming language.

Table 3
A Measurement on QA for the Past

| | **Default Routing** | **CWS Routing** | **CWS Routing with QA merging** |
|---|---|---|---|
| # of QAs | 1 | 3 | 1 (split into 3) |
| Latency (ms) | 4459 | 2887 | 2887 |
| Total # of Hops | 26 | 48 | 28 |

In addition to queries for the past, QAs are used for queries for the future. Figure 14 shows the number of agents (sensor data carried by EAs and QAs) from 3x3 nodes in the center 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.

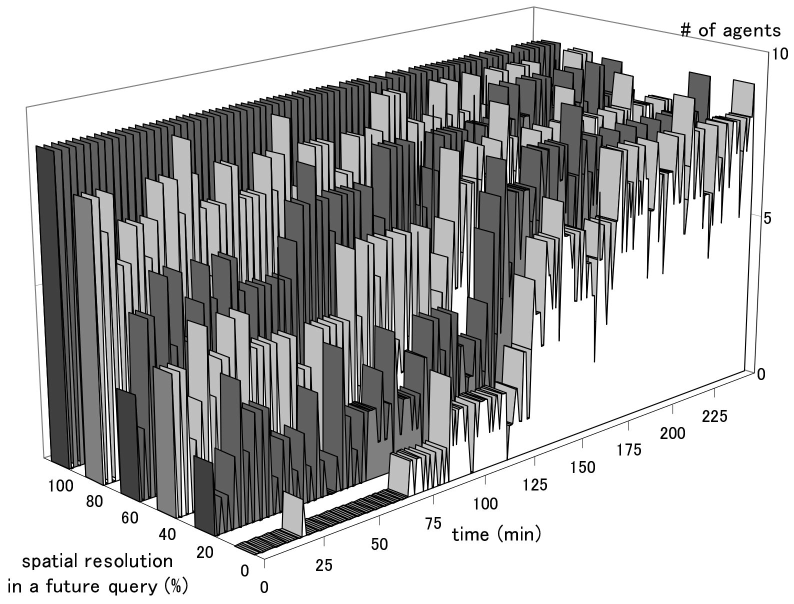


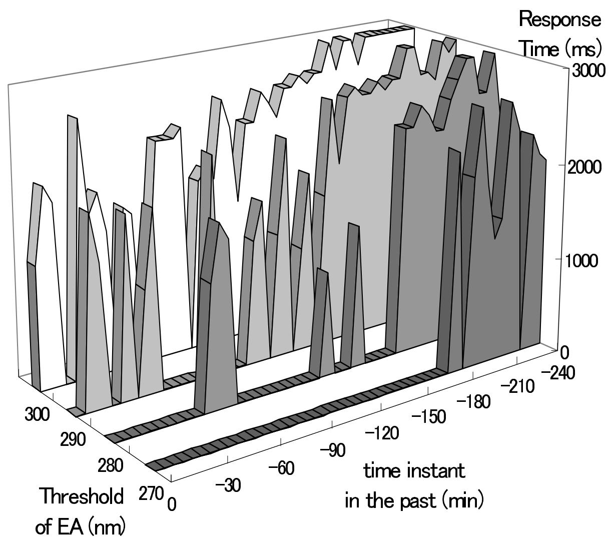
Figure 14. The Number of Sensor Data Received by the Base Station with a Future Query

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 14, spikes appear every three minutes, and they correspond transmitted QAs. This way, a future query in Chronus allows collecting sensor data to satisfy specified spatiotemporal resolutions even if there is no event.

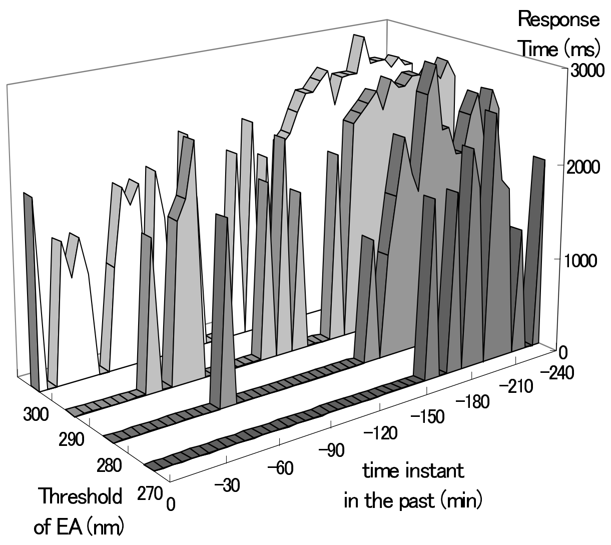
### 6.3. Data Collection in the Past

This section describes differences in response time of data queries that retrieve data in the past from nodes in 3x3 nodes in the center of the WSN with 100% spatial-resolution (15(a) and 15(b)). An application executes data queries, `get_data()` method call with three minutes tolerable delay, 240 minutes after an oil spill occurs. -30 min means that an application examines the data at 30 minutes before. (at 210 minutes after an oil spill occurs.) If every nodes in the 3x3 area sent EAs to a base station, the response time of data queries is almost zero because STDP can provide enough data to an application, However, if STDP cannot provide enough data, Chronus runtime automatically dispatches QAs to sensor nodes to collect data. Since QAs use the CWS algorithm defined in Section 5.3 and data query's tolerable delay is three minutes, it takes 2887ms when no data is available in STDB. Also, it takes 1291ms for a QA to visit to only the closest node in the 3x3 area.

Figure 15(a) and 15(b) show response times when EAs works with Gossip Filtering and Local Filtering algorithms, respectively. Response times in Figure 15(b) is shorter than ones in Figure 15(a) since Local Filtering-based EAs send much sensor data, including false positive data, than Gossip Filtering-based EAs as shown in Section 6.1. When an application gives weights to data collection rather than data detection, Local Filtering-based EAs gives positive impact since it can reduce the response time of data queries for the past. When an application gives weights to data detection rather than data collection, Gossip Filtering-based EAs gives positive impact since it reduces the number of false positive data and consumes less battery power. As the results shown, Chronus allows developers to flexibly tune their applications through the use of micro-programming languages.



(a) With Gossip Filtering-based EAs



(b) With Local Filtering-based EAs

Figure 15. Response Time of Data Queries

## 6.4. Lines of Code

This section compares lines of code (LOC) of applications in Chronus and nesC. A number of WSN applications are currently implemented in nesC [23], a dialect of the C language, and deployed on the TinyOS operating system [24], which provides low-level libraries for basic functionalities such as sensor reading and node-to-node communication. nesC and TinyOS hide hardware-level details; however, they do not help developers to rapidly implement their applications because applications in nesC are required to handle low-level mechanisms (e.g., memory management and routing)

Listing 18 is a Chronus code to collect data from sensor nodes by leveraging QAs. The code hides 1) which routes QAs use to move in WSNs and 2) how each node routes QAs. As described in Section 5.3, Chronus allows reusing micro-programs by leveraging attribute oriented technique. Listing 18 hides 33 lines of code defined in Listing 14. Moreover, application developers are required to implement nesC code deployed on each sensor node to route QAs. Listing 19 shows a fragment of the code. `ReceiveMsg.receive()` is invoked when a node receives a QA (Line 9). After that, a code collects sensor data in the past, append the data to a QA, and route the QA to the next node. In addition to a code in Listing 19, a code for configure modules, e.g., bind the `QARouting` module with a module that implements `ReceiveMsg` interface, is required to be implemented. The total LOC of nesC code is approximately 70.

Listing 18: Chronus code to collect data using QAs

```
# @CWS_ROUTING
sp = Spacetime.new( GLOBALSPACE, Period.new( Hr -1, Min 30 ) )
spaces = sp.get_spaces_every( Min 10, Sec 30, 100 )
```

Listing 19: nesC version QA Routing Module

```
1  module QARouting {
2    uses {
3      interface ReceiveMsg;
4      interface SendMsg;
5      interface StdControl;
6  } }
7  implementation {
8    // invoked once a node receive a QA
9    event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
10     AgentMsg* agent = (AgentMsg*)m->data;
11
12     // collect sensor data from the current node
13     if(agent->isReturning == FALSE &&
14        agent->startCollectionIndex >= agent->numOfHops){
15       uint16_t data = getDataAt(agent->data_timestamp);
16       append(agent, TOS_LOCAL_ADDRESS, data);
17     }
18
19     // check if the current node is the last one to visit
20     if( getLastNode(agent) == TOS_LOCAL_ADDRESS )
21       agent->isReturning = TRUE;
22
23     // go to the next node
24     agent->numOfHops++;
25     uint16_t nextNode = call getNextNode(agent);
26     if(call Send.send(nextNode, sizeof(AgentMsg), &m) != SUCCESS){
27       return FAIL;
28     }
29     return SUCCESS;
30   }
31
32   // get sensor data in the past from a log
```

```
  static uint16_t getDataAt(uint16_t timestamp){...}
  // add a new data to an agent
  static uint16_t append(AgentMsg* agent, uint16_t addr, uint16_t data){...}
  // get the last node to visit
  static uint16_t getLastNode(AgentMsg* agent){...}
  // get the next node to visit
  static uint16_t getNextNode(AgentMsg* agent){...}
}
```

Chronus micro-programs also hides the details of applications running on each sensor node as well as macro-programs does. Listing 20 shows a fragment of nesC version of Local-Filtering algorithm. `ADC.dataReady()` is invoked right after an Analog-Digital Converter module provides sensor data to an application. If the data exceeds a certain threshold (Line 5), it calls `SendData()` to send the data to a base station (Line 7). The LOC of Chronus version of Local-Filtering algorithm is only three (Listing 9) while one of nesC version is 20.

Listing 20: nesC version Local-Filtering Algorithm

```
 1  // invoked once an AD converter is ready
 2  async event result_t ADC.dataReady(uint16_t data){
 3    atomic {
 4      if (!gfSendBusy) {
 5        gfSendBusy = TRUE;
 6        if(data > SENSOR_READING_THRESHOLD) {
 7          post SendData( data );
 8    } } }
 9    return SUCCESS;
10  }
11
12  task void SendData(uint16_t data){
13    EventAgentMsg *pReading;
14    if (pReading = (EventAgentMsg *)call Send.getBuffer(&gMsgBuffer,&Len)) {
15      pReading->type = F_SPECTRUM;
16      pReading->parentaddr = call RouteControl.getParent();
17      pReading->reading = data;
18      // send a message to a base station
19      if ((call Send.send(&gMsgBuffer,sizeof(EventAgentMsg))) != SUCCESS){
20        atomic gfSendBusy = FALSE;
21      }
22  } }
```

As the results shown, Chronus reduces LOC of WSN applications significantly and allows developers to implement their applications rapidly.

## 6.5. Memory Footprint

Table 4 shows the memory footprint of micro-programs deployed on a sensor node. The total footprint counts the memory consumption of TinyOS, Bombilla VM, EA code and QA code. As shown in Table 4, Chronus's micro-programs are lightweight enough to run on a MICA2 node, which has 128 KB memory space. It can operate on a smaller-scale sensor node;for example, TelosB, which has 48 KB memory space.

Table 4
Memory Footprint

| **EA Algorithms** | **Total Footprint (KB)** | **EA Code Footprint (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 a set of extensions to the authors' previous work [25, 26]. This chapter investigates several extensions to [25]; for example, user-defined operators and microprogramming. This chapter also extends [26] to study complex event specification in macro-programs, streamlined mapping from macro-programs to micro-programs based on the notion of attribute-oriented programming, and extra routing optimization for QAs. Moreover, this chapter provides extended simulation results to evaluate the performance and resource consumption of WSN applications built with Chronus.

Pleiades [27] and Snlog [28] are the languages for spatial macroprogramming. They provide programming abstractions to describe spatial relationships and data aggregation operations across nodes. Event detection can be expressed without specifying the low-level details of node-to-node communication and data aggregation. However, these languages require application developers to explicitly write programs to individual nodes. For example, they often need to access the states of individual nodes. In contrast, Chronus allows developers to program event detection to spacetime as a global behavior of each application. Also, Kairos and SNLong do not consider the temporal aspect of sensor data. They do not support data collection for the past; all sensor data are always considered as data collected at the current time frame.

Proto [29], Regiment [30] and Flask [31] are similar to Chronus in that they support in-network data processing and spatiotemporal event detection. While they allow developers to specify arbitrary event detection algorithms, they do not support data collection the notion of spatial and temporal resolutions. Chronus supports data collection for the future and the past in arbitrary spatiotemporal resolutions.

TinyDB [32] and Semantic Stream [33] performs in-network data processing as well as spatiotemporal data collection by extending SQL and Prolog, respectively. They aid to program data collection for the future, but not for the past. Moreover, their expressiveness is too limited to implement data handlers although they are well applicable to specify data queries. Therefore, developers need to learn and use extra languages to implement data handlers. In contrast, Chronus supports spatiotemporal data collection for the future and the past. Its expressiveness is high enough to provide integrated programming abstractions for data queries and data handlers. Also, TinyDB supports event-based data collection that is executed upon a predefined event; however, it does not support event detection on individual nodes.

SPIRE [34] and SwissQM [35] propose SQL-based languages for complex event detection in RFID systems and WSNs, respectively. GEM [36] proposes a Petri Net-based visual language to detect complex events. These languages' expressiveness is too limited to implement event handlers although they are well applicable to define event specifications. Chronus' expressiveness is higher to provide integrated programming abstractions for event specifications and event handlers.

This work is the first attempt to investigate a push-pull hybrid WSN architecture that performs spatiotemporal event detection and data collection. Most of existing push-pull hybrid WSNs do not address spatiotemporal aspects of sensor data [37–39]. They also assume static network structures and topologies (e.g., star and grid topologies). Therefore, data collection can be fragile against node/link failures and node addition/redeployment. In contrast, Chronus can operate in arbitrary network structures and topologies. It can implement failure-resilient queries by having the Chronus server dynamically adjust the migration route that each QA follows.

PRESTO performs push-pull hybrid event detection and data collection in arbitrary network structures and topologies [40]. While it considers the temporal aspect in data queries, it does not consider their spatial aspect. Moreover, it does not support data collection in the future as well as in-network data processing.

## 8. Conclusion

This chapter proposes a new programming paradigm for autonomic WSNs, spatiotemporal macroprogramming. It is designed to reduce the complexity of programming spatiotemporal event detection and data collection. This chapter discusses Chronus' design, implementation, runtime environment and performance implications.

Chronus is currently limited in two dimensional physical space to deploy sensor nodes. It is planned to be extended for using three dimensional physical space (i.e., four dimensional spacetime). Accordingly, new applications will be studied such as three-dimentional building monitoring and atmospheric monitoring.

## REFERENCES

1. I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, A survey of sensor network applications, IEEE Communications Magazine 40 (8) (2002) 102–114.
2. T. Arampatzis, J. Lygeros, S. Manesis, A survey of applications of wireless sensors and wireless sensor networks, in: IEEE International Symposium on Intelligent Control, 2005.
3. D. Estrin, R. Govindan, J. Heidemann, S. Kumar, Next century challenges: Scalable coordination in sensor networks, in: ACM International Conference on Mobile Computing and Networks, 1999.
4. US Coast Guard, Polluting Incident Compendium: Cumulative Data and Graphics for Oil Spills 1973-2004 (September 2006).
5. L. Siegel, Navy oil spills, http://www.cpeo.org/ (November 1998).
6. J. M. Andrews, 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, 1998.
7. C. Brown, M. Fingas, J. An, Laser fluorosensors: A survey of applications and developments, in: The Arctic and Marine Oil Spill Program Technical Seminar, 2001.
8. R. L. Mowery, H. D. Ladouceur, A. Purdy, Enhancement to the ET-35N Oil Content Monitor: A Model Based on Mie Scattering Theory, Naval Research Laboratory (1997).
9. J. A. Nardella, T. A. Raw, G. H. Stokes, New technology in oil content monitors, Naval Engineers Journal 101 (2) (1989) 48–55.
10. US Environmental Protection Agency, Understanding Oil Spills and Oil Spill Response (1999).
11. M. F. Fingas, C. E. Brown, Review of oil spill remote sensors, in: International Conference on Remote Sensing for Marine and Coastal Environments, 2002.
12. G. Booch, Object-Oriented Analysis and Design with Applications, 2nd Edition, Addison-Wesley, 1993.
13. E. Meijer, P. Drayton, Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, in: ACM SIGPLAN Conference

on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Revival of Dynamic Languages, 2004.

14. J. K. Ousterhout, Scripting: Higher level programming for the 21st century, IEEE Computer 31 (3) (1998) 23–30.
15. M. Mernik, J. Heering, A. Sloane, When and how to develop domain-specific languages, ACM Comp. Surveys 37 (4) (2005) 316–344.
16. J. S. Cuadrado, J. G. Molina, Building domain-specific languages for model-driven development, IEEE Software 24 (5).
17. A. Bryant, A. Catton, K. D. Volder, G. Murphy, Explicit programming, in: International Conference on Aspect-Oriented Software Development, 2002.
18. H. Wada, J. Suzuki, K. Oba, S. Takada, N. Doi, mturnpike: a model-driven framework for domain specific software development, JSSST Computer Software 23 (3) (2006) 158–169.
19. T. W. W. W. Consortium (Ed.), SOAP Version 1.2, 2003.
20. P. Levis, D. Culler, Mate: A tiny virtual machine for sensor networks, in: ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.
21. G. Clarke, J. W. Wright, Scheduling of vehicles from a central depot to a number of delivery points, Operations Research 4 (12) (1964) 568–581.
22. C. Beegle-Krause, General noaa oil modeling environment (gnome): A new spill trajectory model, in: Internation Oil Spill Conference, 2001.
23. D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, D. Culler, The nesc language: A holistic approach to networked embedded systems, in: ACM Conference on Programming Language Design and Implementation, 2003.
24. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, K. S. J. Pister, System architecture directions for networked sensors, in: ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.
25. H. Wada, P. Boonma, J. Suzuki, Macroprogramming spatio-temporal event detection and data collection in wireless sensor networks: An implementation and evaluation study, in: Hawaii International Conference on System Sciences, 2008.
26. H. Wada, P. Boonma, J. Suzuki, A spacetime oriented macro programming paradigm for push-pull hybrid sensor networking, in: IEEE ICCC, Workshop on Advanced Networking and Communications, 2007.
27. N. Kothari, R. Gummadi, T. Millstein, R. Govindan, Reliable and efficient programming abstractions for wireless sensor networks, in: SIGPLAN Conference on Programming Language Design and Implementation, 2007.
28. D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, I. Stoica, The design and implementation of a declarative sensor network system, in: ACM Conference on Embedded networked Sensor Systems, 2007.
29. J. Beal, J. Bachrach, nfrastructure for engineered emergence on sensor/actuator networks, IEEE Intelligent Systems 21 (2) (2006) 10–19.
30. R. Newton, G. Morrisett, M. Welsh, The regiment macroprogramming system, in: International Conference on Information Processing in Sensor Networks, 2007.
31. G. Mainland, G. Morrisett, M. Welsh, Flask: Staged functional programming for sensor networks, in: ACM SIGPLAN International Conference on Functional Programming, 2008.
32. S. Madden, M. Franklin, J. Hellerstein, W. Hong, Tinydb: An acqusitional query processing

system for sensor networks, ACM Transactions on Database Systems 30 (1) (2005) 122–173.
33. K. Whitehouse, J. Liu, F. Zhao, Semantic streams: a framework for composable inference over sensor data, in: European Workshop on Wireless Sensor Networks, 2006.
34. R. Cocci, Y. Diao, P. Shenoy, Spire: Scalable processing of rfid event streams, in: RFID Academic Convocation, 2007.
35. R. Müller, G. Alonso, D. Kossmann, Swissqm: Next generation data processing in sensor networks, in: Conference on Innovative Data Systems Research, 2007.
36. B. Jiao, S. Son, J. Stankovic, Gem: Generic event service middleware for wireless sensor networks, in: International Workshop on Networked Sensing Systems, 2005.
37. W. Liu, Y. Zhang, W. Lou, Y. Fang, Managing wireless sensor networks with supply chain strategy, in: International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks, 2004.
38. W. C. Lee, M. Wu, J. Xu, X. Tang, Monitoring top-k query in wireless sensor networks, in: IEEE International Conference on Data Engineering, 2006.
39. S. Kapadia, B. Krishnamachari, Comparative analysis of push-pull query strategies for wireless sensor networks, in: International Conference on Distributed Computing in Sensor Systems, 2006.
40. M. Li, D. Ganesan, P. Shenoy, Presto: Feedback-driven data management in sensor networks, in: ACM/USENIX Symposium on Networked Systems Design and Impl., 2006.