Middleware Support for Pluggable Non-functional Properties in Wireless Sensor Networks

Pruet Boonma and Junichi Suzuki
Department of Computer Science
University of Massachusetts, Boston
{pruet, jxs}@cs.umb.edu

Abstract

Wireless sensor networks (WSN) imposes stringent constraints on efficiency, memory footprint and power consumption. Since the need to satisfy these constraints often results in tightly coupled designs, WSN applications tend to be inflexible; it is hard to flexibly reuse, introduce, customize and replace various non-functional properties (e.g., data routing, concurrency, data aggregation and event filtering) for developing and maintaining WSN applications. In order to address this issue, this paper proposes the TinyDDS middleware, which decouples various non-functional properties from WSN applications and allows those applications to flexibly reuse and transparently configure non-functional properties according to their own requirements. Without breaking the generic architecture of TinyDDS, the proposed pluggable framework allows WSN applications to have fine-grained control over non-functional properties and specialize in their own requirements. Currently, TinyDDS supports two types of non-functional properties: application-level and middleware-level non-functional properties.

1. Introduction

Wireless sensor networks (WSNs) require per-node embedded software that imposes stringent constraints on efficiency, memory footprint and power consumption. Since the need to satisfy these constraints often results in tightly coupled designs, WSN applications tend to be inflexible; it is hard to flexibly reuse, introduce, customize and replace various non-functional properties (e.g., data routing, concurrency, data aggregation and event filtering) for developing and maintaining WSN applications. This can substantially increase the effort and cost of WSN application development and maintenance.

In order to address this issue, this paper investigates TinyDDS, which is an open-source[1] and standards-based middleware for resource and performance sensitive WSNs. Compliant with the Data Distribution Service (DDS) standard specification [6], TinyDDS is designed and implemented generic enough to aid in developing a wide range of event detection and data collection applications.

The core of TinyDDS is a pluggable framework that decouples various non-functional properties from WSN applications and allows those applications to flexibly reuse and transparently configure non-functional properties according to their own requirements. For example, an event detection and tracking application may require some in-network event correlation and filtering mechanisms as its non-functional properties in order to eliminate false positive data in the network. A periodical data collection application may require a data aggregation mechanism and power-efficient routing mechanism as its non-functional properties in order to reduce traffic volume and expand the network's lifetime. Without breaking the generic architecture of TinyDDS, the proposed pluggable framework allows WSN applications to have fine-grained control over non-functional properties and specialize in their own requirements. Currently, TinyDDS supports two types of non-functional properties: *application-level* and *middleware-level* non-functional properties.

This paper is organized as follows. Section 2 overviews the DDS specification, and Section 3 describes the design and implementation of TinyDDS. Section 4 discusses how non-functional properties are decoupled from and reusable by applications. Sections 5 and 6 conclude with some discussion on related and future work.

2. An Overview of the OMG DDS Specification

The Data Distribution Service (DSS) is an Object Management Group's standard specification for publish/subscribe middleware in distributed network systems. An application implemented on top of DDS can publish events, i.e. data or control messages, to the network with associated topic and the events are captured by any network

[1] TinyDDS is available at dssg.cs.umb.edu.

978-0-7695-3286-8/08 $25.00 © 2008 IEEE
DOI 10.1109/SERVICES-1.2008.78

nodes subscribe to the topic of that events. According to [6], DDS consists of two levels, a lower Data-Centric Publish-Subscribe (DCPS) level and an optional higher Data Local Reconstruction Layer (DLRL). DCPS is a middleware which allows application that want to sent out events declare their intent to become **Publishers** of particular event's topics. On the other hand, applications that want to access some particular event's topics declare their intent to become **Subscriber**. Each time a Publisher sent out new events of particular topic into this middleware, the middleware propagate the events to all interested Subscribers. DLRL automatically updates events locally from the remote updates and allows the application to access the events 'as if' they were local. In this work, only DCPS is implemented because, given the sensor nodes' limited resources, the overhead of implementing DLRL is not worth its benefit. Figure 1 shows the DDS architecture.

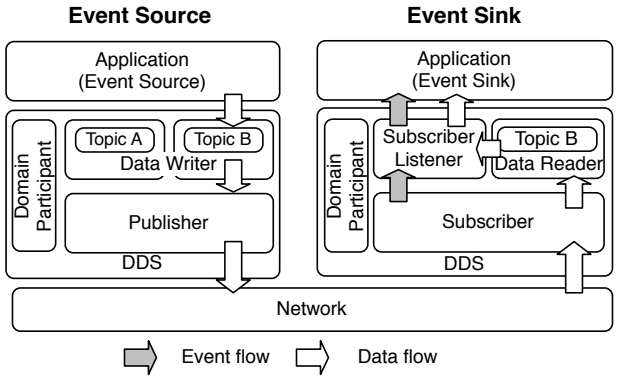

Figure 1. DDS Architecture

From the figure, on the event sink application, single instance of the `DomainPartipant` class is instantiated in each middleware. The instance of this class maintains the list of all other instances initiated in the middleware. Application first instantiate an instance of Publisher from `DomainParticipant`. Then, the application instantiate a set of instances of the `Topic` class depends on how many topic interested by this application. Topic is an identifier that uniquely identifies particular event's content. On the lowest level, an instance of the `Subscriber` class keeps monitoring the lower-level network traffic and capture any events from the network. Then, the `Subscriber` instance checks the topic of captured events and inform the instances of `SubscriberListener` and `DataReader` which are associated with that topic for the availability of the events. Next, `SubscriberListener` informs application about the existence of the events and the application read the events from `SubscriberListener`, which internally read data from `DataReader`. Figure 3 shows the sequence diagram of publication process.

On the event source application, the application pub-

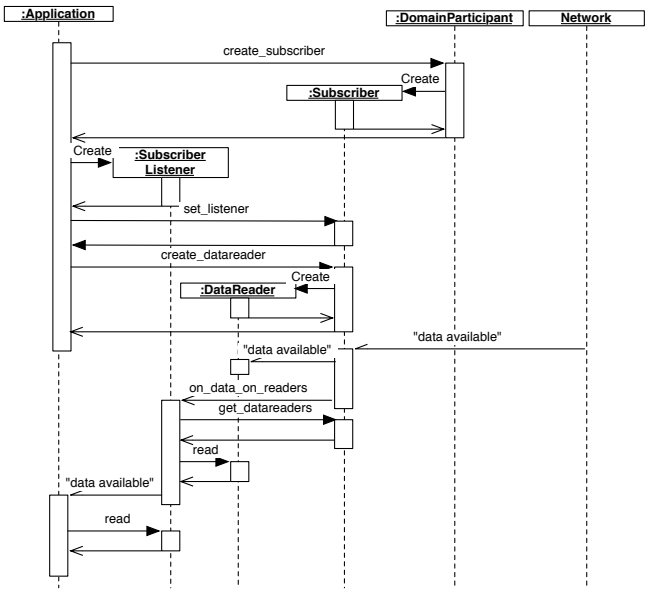

Figure 3. Subscription Process

lishes its events through instances of the `DataWriter` class. Each topic defined in the system has an associated `DataWriter` instance, thus application has to choose appropriate `DataWriter` instance depends on topic of the events it wants to publish. Events from all `DataWriter` are published through the single instance of the `Publisher` class to the lower-level network infrastructure. Figure 4 shows the sequence diagram of publication process.

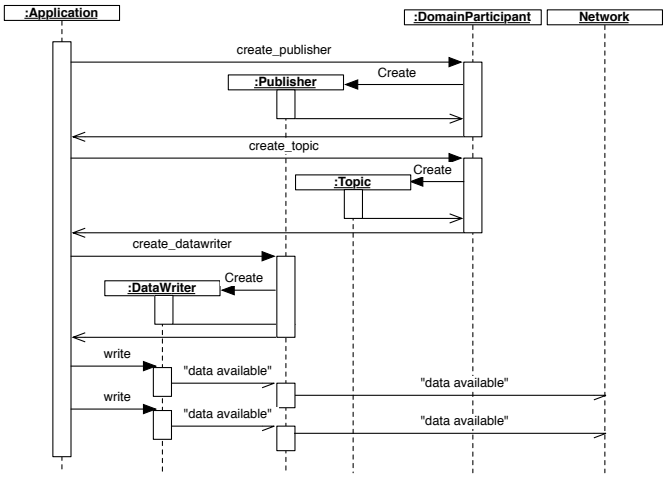

Figure 4. Publication Process

Instead of receiving all events associated with a topic, the application can create a special kind of topic which is an instance of the `ContentFilteredTopic` class. This

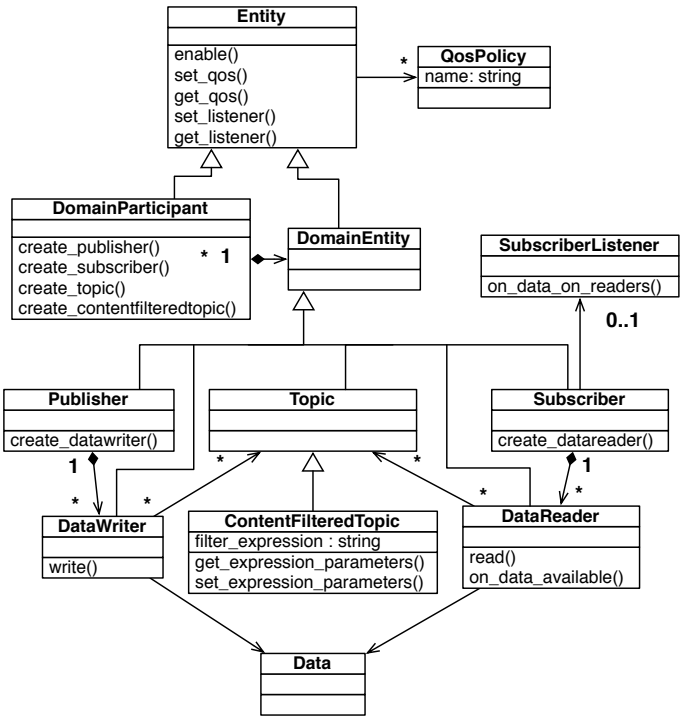

Figure 2. Standard DDS Interfaces

`ContentFilteredTopic` indicates the subscriber does not want to receive all events published under the topic. Rather, it wants to see only the events whose contents satisfy certain criteria. For example, an application may be interested in a topic called Temperature and only when the events of this Temperature topic is in between 100 and 150; therefore, `ContentFilteredTopic` sets the criteria as *Temperature > 100 AND Temperature < 150*. The format of criteria expression is a subset of SQL syntax. Moreover, for each DDS components, such as `Topic`, `DataWriter`, `DataReader`, `Subscriber` or `Publisher`, a QoS policy can be specify to govern their behavior. Application developers can define the QoS policy through an instance of the `QosPolicy` class. Then, `QosPolicy` is attached to the desired component. The DDS middleware tries to meet the QoS policy by adjusting its working parameters. In the other words, QoS policy is a non-functional properties of the application running on DDS middleware.

Figure 2 shows a partial interface diagram of DDS.

3. TinyDDS

Figure 5 shows the TinyDDS components diagram. With respect to TCP/IP reference model, TinyDDS operates in transport layer and work on top of any network layer implementation. TinyDDS follows Layer design pattern [1] by separating different functionalities into different layers. At the top most layer, TinyDDS defines a subset of DDS interfaces to be used by applications. Then, the implementation of those interfaces, as described in section 2, operates on top of an overlay network for event routing. Different routing protocols can be used to implement the overlay network by implementing in the Overlay Event Routing Protocols (OERP) layer. This OERP layer allows application developer to choose appropriate routing protocol to suit their requirements and constraints. For example, in sensor network with very limited memory space sensor nodes, flooding-based routing protocol may be used because it needs minimal memory space to maintain routing table. On the other hand, sensor network which try to minimize the energy consumption of memory rich sensor nodes may use DHT-based routing protocol. By using this OERP layer, TinyDDS frees developers from the limitation of routing algorithm used in network layer which generally depends on sensor node platform such as MicaZ which based on Zigbee protocol stack. The routing protocol in OERP layer utilizes low-level network layer implementation through a transport layer interface called TinyDDS L4 Adaption Layer (L4AL). L4AL allows TinyOS to operates with any network and MAC layer protocol, such as AODV and Zigbee respectively.

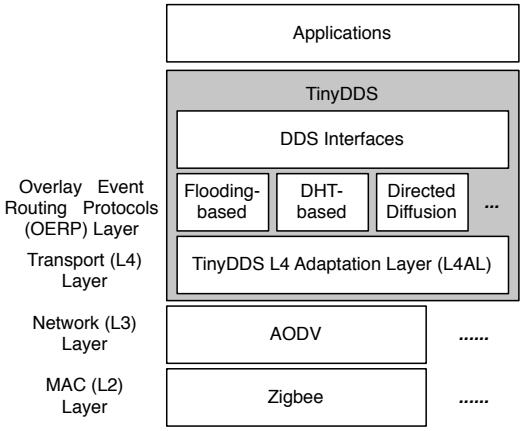

Figure 5. Architectural Components in TinyDDS

3.1. DDS Interfaces

In the top most layer, TinyDDS provides an API for application developers. This API provides a subset of DDS interfaces which is implemented as shown in the interface diagram in figure 2. TinyDDS provides interfaces for creating topics, subscribe to events of topics and publish events for particular topics. For each interfaces, the implementation is provided so application developers do not need to implement those interfaces themselves. In particular, Bridge design pattern [2] is used to hide low-level implementation from application developers. The implementation for the DDS interfaces is written in nesC programming language and optimized for small sensor nodes platform such as MicaZ using several design pattern such as dispatcher is used to handle *Topic* interface. In particular, instead of instantiating different Topic into different nesC components, only single nesC complement is used to handle all topics. Then, application developers can follow the sequence diagram on figure 3 and 4 for subscribing and publishing events respectively. Listing 1 shows a fragment of an event sink application which subscribe to a topic called "TempSensor", i.e., temperature sensor reading.

```
1  Subscriber_t subscriber;
2  Topic_t ts_topic;
3  DataReader_t data_reader;
4  SubscriberListener_t listener;
5  command result_t StdControl.start() {
6   subscriber = call DomainParticipant.create_subscriber();
7   ts_topic = call DomainParticipant.create_topic("TempSensor");
8   listener  = call SubscriberListener.create(ts_topic);
9   call Subscriber.set_listener(listener);
10  data_reader = call Subscriber.create_datareader(
11                                      ts_topic, listener);
12 }
13
14 event ReturnCode_t SubscriberListener.data_available(
15                                          Topic_t topic) {
16   Data event;
17   if(topic == ts_topic) {
18     event = call SubscriberListener.read();
19     // processing event..
20   }
21 }
```

Listing 1. Event Sink Application

From the listing, in the method `StdControl.start()` which is called when the sensor node starts, the application creates an instance of the `Subscriber`, `Topic`, and `DataReader` class and also register an `SubscriberListener`. Then, in the method `SubscriberListener.data_available()`, which is called when the underlying Subscriber receives an event with topic "TempSensor", the application can get that event from `SubscriberListener`.

```
1  Publisher_t publisher;
2  Topic_t topic;
3  DataWriter_t data_writer;
4  Data data;
5  command result_t StdControl.start() {
6   publisher = call DomainParticipant.create_publisher();
7   topic = call DomainParticipant.create_topic("TempSensor");
8   data_writer = call Publisher.create_datawriter(
9                                              publisher , topic);
10  // Get sensor reading, and put to data variable
11  call DataWriter.write(data_writer, data);
12 }
```

Listing 2. Event Source Application

Listing 2 shows a fragment of an event source application. From the source code in the method `StdControl.start()`, the application create an instance of the `Publisher`, `Topic`, and `DataReader` class. Then, it publishes a data through the `DataWriter`.

3.2 Overlay Event Routing Protocols (OERP)

This OERP layer provides an overlay network over sensor network's physical ad-hoc networks. The overlay network is used for transporting published events, i.e. sensor data, to all nodes who subscribes to the events. The published event is routed to each subscribers according to the routing protocols deployed within OERP layer. Application developers can specify the deployed routing protocols to suit their need. The OERP layer encapsulates the overlay network algorithm and implementation from DDS interfaces and the lower level physical network. Routing protocols in OERP layer work with lower-level network protocol through the L4AL. In the other words, routing protocols can be seen as a non-functional properties of TinyDDS which can be deployed to meet application developers' non-functional requirements. For example, application developers who want to reduce the price and size of sensor nodes by using small-memory sensor nodes may choose to use flooding-based routing protocol which will use very small memory space. The routing protocols used in this OERP framework are developed by library developers and can be

used in any TinyDDS-based applications.

3.3 TinyDDS L4 Adaptation Layer (L4AL)

To access to low level physical network, the routing protocols in OERP make use of low level physical network through a network abstract layer called TinyDDS L4 Adaptation Layer (L4AL). This L4AL utilize Bridge design pattern to separate the real low level physical network implementation from the higher level overlay network. Thus, TinyDDS can be portable among different sensor platform. In particular, L4AL provides an interface to access physical network functions such as how to get the list of neighboring nodes, how to get the link quality to each neighboring node and also how to send/receive data to/from particular nodes in the network. These functions are used by the routing protocols on the OERP layer and implemented by the Network Layer implementation. Internally, L4AL contains a set of tables that maintains the information of network, such as neighbor list and link quality, and a set of event queues. There are two types of event queues, incoming queues and outgoing queues. The events submitted from OERP for sending out to physical network is put to the end of outgoing queue while the events collected from physical network are put to the end of incoming queue, waiting to be processed by the routing protocol in OERP.

3.4. Application Development with TinyDDS

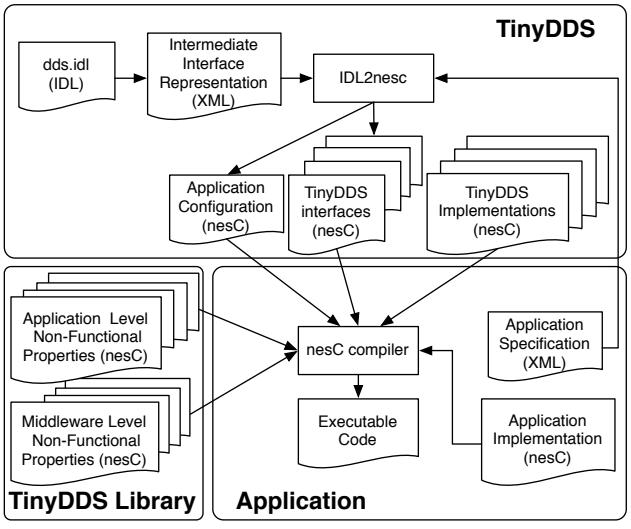

Figure 6. Application Development Model

Figure 6 shows the development model of an TinyDDS application. There are three main components of the development model, TinyDDS middleware, TinyDDS Library and the application. The TinyDDS middleware comprises of two parts, the DDS interfaces definition and the TinyDDS implementation of the interfaces. The DDS interfaces definition is directly generated from the *dds.idl*, which is the official DDS interfaces definition in IDL format from OMG. The *dds.idl* is first converted into XML format. Then, *IDL2nesc* converts the DDS interfaces definition from XML format to *TinyDDS interfaces* and *Application Configuration*. The Application Configuration follows Facade design pattern [2] and describes how to connect each interfaces and implementation together. IDL2nesc also uses an *Application Specification*, written in XML, in order to generates appropriate Application Configuration, for example, Application Specification specifies which routing protocol will be used in OERP layer, then Application Configuration connects the implementation of the routing protocol into OERP interface.

The second components is the *TinyDDS Library*. TinyDDS Library consists of two non-function properties implementation, namely, application-level and middleware-level non-functional properties. The application-level non-function properties provides a set of services which can be used by application, such as data aggregation and event detection. The middleware-level non-function properties provide the services inside the middleware, for example, routing protocols in OERP layer. Library developer develops these functionality in the TinyDDS Library and the TinyDDS Library can be used in any application on any hardware platform.

The third components is the *Application* itself. The Application consists of two parts, the Application Specification which is used by the IDL2nesc compiler and the *Application Implementation*. The Application Implementation is developed by application developer and perform a certain task such as data collection and event detection. The examples of Application Implementation are shown on the Listing 1 and 2. The Application Specification describes the overview of the application, for example, what is the target platform, or which routing protocol will be used in OERP layer. Listing 3 shows an example of the Application Specification.

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <configuration platform="micaz" threading="per-event">
3   <includes>
4    <header name="BaseUART" />
5    <header name="DDS_utils" />
6    <component name="DDS_DataAggregation" />
7    <component name="LedsC" />
8    <component name="Flooding" />
9   </includes>
10  <implementations>
11   <implementation component="Flooding" interface="OERP" />
12  </implementations>
13  <connections>
14   <connection from="Main.StdControl" to="*" />
15   <connection from="Application.DataAggregation"
16               to="DDS_DataAggragation" />
17   <connection from="Application.Leds" to="LedsC" />
```

```
18   </connections>
19  </configuration>
```

Listing 3. Application Specification

From the listing, there are three parts in this XML files. The first part, inside the <**includes**> tag, describes the list of external component and header files will be used in this application. For example, in this application,*DDS_DataAggragation*, *LedsC*, and *Flooding* components are used in the application. Then, the second part, inside the <**implementations**> maps which component will be used with OERP interface. In particular, this application uses *Flooding* as its event routing protocol in OERP layer. Then, the final part, inside the <**connections**>, maps connections between component. For example, *DS_DataAggration* and *LedsC* will be used in the application implementation. Notice that, *Application* is the default application interface which will be mapped to the application implementation from application developer. The <**configuration**> tag controls working properties of the middleware, for example, the target platform that this middleware will operate on and the threading model used by this middleware.

Then, the nesC compiler combines the Application Configuration, TinyDDS Interfaces, TinyDDS Implementations, Application Implementations and the implementation from TinyDDS Library into target executable code.

4. Non-Functional Properties of TinyDDS

To ease the application development on sensor nodes, TinyDDS provides non-functional properties both on application and middleware level collectively as a library, called TinyDDS library in the figure 6. The application-level non-functional properties accelerates the application development process by providing frequently used non-functional properties such as data aggregation and event detection. Thus, application developers can focus more on their application functionality, e.g. how to interpret and process those data and event. Moreover, utilizing non-functional properties can reduce the application complexity and thus improve the maintainability. On the other hand, middleware-level non-functional properties allows application developers to adjust the behavior of the middleware to suit their need and constraints, i.e., choosing event routing protocol which suite the application or specify the QoS of each middleware components. In addition, TinyDDS library is designed to be portable and can be used by many TinyDDS based application. Therefore, by using both application and middleware-level non-functional properties, application developers can gain better reusability, maintainability, composability and performance.

4.1. Application-Level Non-Functional Properties

In the application level, non-functional properties in TinyDDS helps application developers to rapidly develop their applications. Application developers can use application-level middleware services such as data aggregation instead of subscribing/publishing directly to TinyDDS middleware. Data aggregation collects and process data from sensor network and provides processed data to application. Processing operators supports by data aggregation are, for example, summation, average, maximum, and minimum. The data aggregation component is plugable, application developers can include this in their application using Application Specification (see Section 3.4). In addition, library developers can develop any non-functional components, such as network security and persistence storage, which are reusable and plugable to all TinyDDS applications.

```
1  Subscriber_t subscriber;
2  Topic_t ts_topic;
3  DataAggregator_t data_aggregrator;
4  SubscriberListener_t listener;
5  command result_t StdControl.start() {
6   subscriber = call DomainParticipant.create_subscriber();
7   ts_topic = call DomainParticipant.create_topic("TempSensor");
8   listener  = call SubscriberListener.create(ts_topic);
9   call Subscriber.set_listener(listener);
10  data_aggregrator = call Subscriber.create_data_aggregrator(
11                              ts_topic, AVERAGE, listener);
12 }
13
14 event ReturnCode_t SubscriberListener.data_available(
15                                              Topic_t topic) {
16   Data data;
17   if (topic == ts_topic) {
18     data = call SubscriberListener.read();
19     // processing aggregated data..
20   }
21 }
```

Listing 4. Event Sink Application using Data Aggregation

Listing 4 shows a fragment of an event sink application using data aggregation. The application is similar to the application in the listing 1 except that it creates an instance of the class `DataAggregrator` instead of `DataReader`. Then, when new data is available, the TinyDDS middleware informs the application using data_available event and provide aggregated data, in this case, average temperature sensor, to the application.

```
1  Subscriber_t subscriber;
2  ContentFilteredTopic_t ts_topic;
3  DataAggregator_t data_aggregrator;
4  SubscriberListener_t listener;
5  command result_t StdControl.start() {
6   subscriber = call DomainParticipant.create_subscriber();
7   ts_topic = call DomainParticipant.create_topic("TempSensor");
8   call ContentFilteredTopic.set_expression_parameters(
9            topic, "TempSensor > 100");
10  listener  = call SubscriberListener.create(ts_topic);
11  call Subscriber.set_listener(listener);
12  data_aggregrator = call Subscriber.create_data_aggregrator(
13                              ts_topic, UNIQUE, listener);
14 }
```

```
15
16 event ReturnCode_t SubscriberListener.data_available(
17                                         Topic_t topic) {
18   Data node_count;
19   if(topic == ts_topic) {
20     node_count = call SubscriberListener.read();
21     if(node_count > 50) {
22     // The event is happening
23     }
24   }
25 }
```

Listing 5. Event Detection Application using Event Filtering

Listing 5 shows a fragment of an event detection application which uses data aggregation and event filtering non-functional properties of TinyDDS. An instance of the class `ContentFilteredTopic` is created instead of `Topic`. Then, the filtered expression is set to the topic, in this example, the application is interested only temperature sensor data which has value greater than 100. Then, a `DataAggregrator` is created to aggregate data using UNIQUE operator. The UNIQUE operator distinctively count the number of sensor nodes who publish the data in this topic. Therefore, in the data_available event `SubscriberListener` returns the number of nodes who already published the data, instead of the data value itself.

4.2. Middleware-Level Non-Functional Properties

TinyDDS supports three non-functional properties in the middleware level, the pluggable routing protocols in OERP layers (see Section 3.2 and 3.3), concurrency, and the QoS policy. Application developers can specify the concurrency model used in TinyDDS. In particular, TinyDDS supports two concurrency model, thread-per-event and thread-per-event-topic. In the thread-per-event model, TinyDDS creates a thread, e.g., a Task in TinyOS, for each event submitted from application or collected from the network. This model gives same priority for each event. In the other words, TinyDDS has only two event queues, one for outgoing events and the other for incoming events, both of them working in first-come-first-serve fashion. On the other hand, the thread-per-event-topic allows application developer to specify different priority for each event topic. In particular, TinyDDS creates two message queues for each event topic, one for incoming events and the other for outgoing events. Then, TinyDDS processes the event queues based on priority of each event topic, for example, TinyDDS publishes events with high priority topic more frequent than one with low priority topic. The concurrency model and its working parameters, e.g. topic priority, can be specified in Application Specification (see listing 3).

For the QoS policy, TinyDDS utilizes some of the QoS model of DDS, such as latency budget and reliability. Latency budget QoS policy specifies the maximum accepted latency from the time the event is published until the event is available to the destination subscribers. Listing 6 shows a fragment of an application using latency budget QoS policy.

```
1  ...
2  Publisher_t publisher;
3  Topic_t topic;
4  DataWriter_t data_writer;
5  QosPolicy_t qos;
6  Data data;
7  command result_t StdControl.start() {
8   publisher = call DomainParticipant.create_publisher();
9   qos = call QosPolicy.create_latency_budget_qos(100);
10  topic = call DomainParticipant.create_topic("TempSensor");
11  call Topic.set_qos(topic, qos);
12  data_writer = call Publisher.create_datawriter(
13                                          publisher , topic);
14  // Get sensor reading, and put to data variable
15  call DataWriter.write(data_writer, data);
16 }
```

Listing 6. Latency Budget QoS Parameters Setting

From the listing, the application creates a latency budget QoS policy with 100 ms constraint. Then, the QoS policy is applied to the `Topic` instance which imply that the data in this topic should be delivered with less than 100 ms latency. TinyDDS uses mechanisms in L4AL to satisfy the QoS policies. For example, to satisfy the latency budget QoS policy, TinyDDS rearranges the order of event in the event queues such that the event with has a high chance to break the QoS policy, e.g., event's actual latency is already very close to the desired latency, will be published earlier than the event which has the less chance to break the QoS policy, e.g., event's actual latency is very far from the desired latency.

The reliability QoS policy indicates the level of data transmission reliability provides by TinyDDS. In particular, TinyDDS supports two reliability model, RELIABLE and BEST_EFFORT. When reliability QoS policy is set to RELIABLE, TinyDDS attempts to deliver all events. The missed events are retransmitted until the number of transmission is greater than a threshold or the transmission is success. On the other hand, when reliability QoS policy is set to BEST_EFFORT, TinyDDS sends out each event only once and relies on MAC layer for succeeding the transmission. Listing 7 shows a fragment of an application using reliable QoS policy.

```
1  ...
2  Publisher_t publisher;
3  Topic_t topic;
4  DataWriter_t data_writer;
5  QosPolicy_t qos;
6  Data data;
7  command result_t StdControl.start() {
8   publisher = call DomainParticipant.create_publisher();
9   qos = call QosPolicy.create_reliability_qos(
10                              BEST_EFFORT_RELIABILITY_QOS);
11  topic = call DomainParticipant.create_topic("TempSensor");
12  call Topic.set_qos(topic, qos);
13  data_writer = call Publisher.create_datawriter(
14                                          publisher , topic);
15  // Get sensor reading, and put to data variable
16  call DataWriter.write(data_writer, data);
```

```
17 }
```

Listing 7. Reliability QoS Parameters Setting

5. Related Work

TinyDDS operates atop TinyOS on each sensor node. TinyOS [3] is an operating system designed for sensor nodes with small memory footprint. TinyOS provides a component-based architecture which allows application developers to develop application using a C-like programming language, called nesC. Because TinyOS is positioned as a generic platform for all kind of sensor network application, so application developers have to handle all low-level task such as memory allocation and data communication. Unlike TinyOS, TinyDDS allows application developers to focus on their task at hand, i.e., how to process sensor data, and hides all low level detail from developers. For example, a simple data collecting application called Surge in TinyOS contains about 280 lines of nesC code. Similar application can be done in TinyDDS with only 30 lines of nesC code. Moreover, TinyDDS provides an easy configuration systems through a single XML file which allows application developers to adjust the characteristic of the TinyDDS. Thus, application developers do not need to manually fine tune their application code to meet their requirement.

Several research efforts have proposed pub/sub middleware to sensor networks. In [7], a pub/sub middleware for sensor networks called Mires is proposed. Mires, operates on top of TinyOS, allows application to subscribe and publish to data with particular topic. On top of TinyOS, Mires provides a pub/sub service bus and high-level services such as routing and data aggregation. In contrast with TinyDDS, Mires provides only application-level non-functional properties such as data aggregation but it does not provide middleware-level non-functional properties such as QoS or plugable event routing. Thus, application developers can not tailor the middleware to fit their application requirement. [4] proposes a pub/sub middleware, called SMC, for managing body-sensor networks. The middleware operates on a powerful node with Linux and Java VM and consists of an event bus and several components running on the event bus such as service discovery, policy management and resource management. The application is written in form of a policy, i.e., a set of rules to be performed for a particular event. Each component communicate with the other components and the other sensor nodes through the event bus using pub/sub scheme. Compared with TinyDDS, SMC does not provide any middleware-level non-functional properties such as QoS. Moreover, SMC supports only constrained-based subscription, i.e., subscribers have to create a filter, e.g. temperature ¿ 100, in order to receiving event. TinyDDS, on the other hand, supports both topic-based and constrained-based subscription. DSWare is a pub/sub middleware designed primarily for event detection application [5]. In contrast with TinyDDS, DSWare does not provide any application-level non-functional properties, application developers need to implement their application from the ground up based on the pub/sub model provides by DSWare. Similar to TinyDDS, DSWare provides some middleware-level non-functional properties such as data storage or caching (TinyDDS provides different type of services); however, application developers cannot fine tune the characteristic of the middleware, neither by configuration file or adjusting QoS parameters.

6. Conclusion

This paper describes a pluggable framework in the TinyDDS middleware, which decouples various non-functional properties from WSN applications and allows those applications to flexibly reuse and transparently configure non-functional properties (e.g., data routing, concurrency, data aggregation and event filtering) according to their own requirements. TinyDDS supports two types of non-functional properties: *application-level* and *middleware-level* non-functional properties. This paper describes the design and implementation of TinyDDS, and discusses how non-functional properties are decoupled from and reusable by applications.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* Wiley and Sons, 1996.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proc. of ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[4] S. L. Keoh, N. Dulay, E. Lupu, K. Twidle, A. E. Schaeffer-Filho, M. Sloman, S. Heeps, S. Strowes, and J. Sventek. Self-managed cell: A middleware for managing body-sensor networks. In *Proc. of Int'l Conf. on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2007.

[5] S. Li, Y. Lin, S. H. Son, J. A. Stankovic, and Y. Wei. Event detection services using data service middleware in distributed sensor networks. *Springer Telecomm. Systems*, 26(2-4), 2004.

[6] Object Management Group. Data Distribution Service (DDS) for real-time systems, v1.2, 2007. `http://www.omg.org/technology/documents/formal/data_distribution.htm`.

[7] E. Souto, G. G. aes, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner. Mires: a publish/subscribe middleware for sensor networks. *Springer Personal Ubiquitous Computing*, 10, 2005.