TinyDDS: An Interoperable and Configurable Publish/Subscribe Middleware for Wireless Sensor Networks

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

ABSTRACT

Due to stringent constraints in memory footprint, processing efficiency and power consumption, traditional wireless sensor networks (WSNs) face two key issues: (1) a lack of interoperability with access networks and (2) a lack of flexibility to customize non-functional properties such as event filtering, data aggregation and routing. In order to address these issues, this chapter investigates interoperable publish/subscribe middleware for WSNs. The proposed middleware, called TinyDDS, enables the interoperability between WSNs and access networks by providing programming language interoperability and protocol interoperability based on the standard Data Distribution Service (DDS) specification. Moreover, TinyDDS provides a pluggable framework that allows WSN applications to have fine-grained control over application-level and middleware-level non-functional properties. Simulation and empirical evaluation results demonstrate that TinyDDS is lightweight and efficient on the TinyOS and SunSPOT platforms. The results also show that TinyDDS simplifies the development of publish/subscribe WSN applications.

INTRODUCTION

Wireless sensor networks (WSNs) have been used to detect events and/or collect data in various domains such as environmental observation, structural health monitoring, human health monitoring, inventory tracking, home/office automation and military surveillance. Due to their deeply-embedded pervasive nature, WSNs have a potential to revolutionize the way that humans understand and construct complex natural/physical systems (Estrin et al., 1999).

A WSN application requires per-node embedded software that imposes stringent constraints in memory footprint, processing efficiency and power consumption. In order to satisfy these constraints, traditional WSN applications often result in *vertically integrated* and *tightly coupled* designs. Vertically integrated designs make WSN applications less interoperable. For example, most of traditional WSNs lack interoperability with access networks, which allow human users to connect to WSNs and perform information retrieval such as data collection and event detection (Henricksen & Robinson, 2006; Romer

et al., 2002; Hadim & Mohamed, 2006). Despite the interoperability can foster the practicality and production deployment of WSNs, they have been investigated and designed separately from access networks. As a result, it is often ad-hoc, expensive and error-prone to build a gateway node, which is responsible for protocol bridging and data conversion between WSNs and access networks. Currently, gateways need to be rebuilt from scratch when WSNs and access networks use different programming languages and protocols.

Tightly coupled designs make WSN applications less flexible. In WSN applications, it is hard to flexibly introduce, reuse, customize and replace various non-functional properties such as event correlation, event filtering, data aggregation and routing policies. Currently, changes in non-functional properties require substantial re-designs and re-programming of WSN applications. As a result, the productivity of WSN application development remains low, and the cost of application maintenance remains high.

In order to address the aforementioned interoperability and flexibility issues, this chapter investigates interoperable publish/subscribe communication with TinyDDS, which is open-source[1], standards-based and configurable middleware for WSNs. It is designed and implemented generic enough to aid in developing a wide range of event detection and data collection applications. Compliant with Object Management Group (OMG)'s standard Data Distribution Service (DDS) specification (Object Management Group, 2007), TinyDDS provides two types of interoperability: *programming language interoperability* and *protocol interoperability*.

Programming language interoperability is the ability of TinyDDS to interoperate applications written in different programming languages. TinyDDS implements a set of standard DDS APIs in nesC[2] (Gay et al., 2003) and Java Micro Edition (Simon & Cifuentes, 2005) by providing mappings of the OMG Interface Definition Language (IDL) (Object Management Group, 2007) to the two languages. TinyDDS' nesC version operates on the TinyOS platform (Levis, et al., 2005), and its Java version operates on the SunSPOT platform (Simon & Cifuentes, 2005). This allows different applications to use different languages with the same DDS APIs. For example, an access network application (or end-user application) may be implemented with Java or JavaScript, while a WSN application may be implemented with nesC or Java. Application developers do not have to learn/use different APIs for different applications. This can significantly improve their productivity in application development.

Protocol interoperability is the ability of TinyDDS to interoperate WSNs and access networks built with different MAC (L2), routing (L3) and transport (L4) protocols. TinyDDS implements a session (L5) protocol, called TinyGIOP, which is a subset of the OMG General Inter-ORB Protocol (GIOP) (Object Management Group, 2007). Similar to GIOP, TinyGIOP is independent from underlying L2 to L4 protocols. It transmits data formatted with TinyCDR, which is a subset of the OMG Common Data Representation (CDR) (Object Management Group, 2007). CDR defines the standard binary representations of IDL data types. Taking advantage of TinyGIOP and TinyCDR, TinyDDS allows gateway nodes to be reusable to bridge various WSNs and access networks even if the two networks use different L2, L3 and L4 protocols. This way, it is intended to reduce the costs (time and labor) to build and maintain gateways. TinyDDS is the first DDS implementation for the TinyOS and SunSPOT platforms.

TinyDDS addresses the flexibility issue described earlier by providing a pluggable framework that decouples various non-functional properties from WSN applications. The framework allows WSN applications to flexibly reuse and configure non-functional properties according to their requirements. For

[1]TinyDDS is available at dssg.cs.umb.edu.
[2] nesC is a dialect of the C language for WSNs.

example, an event detection application may require in-network event correlation and filtering as its non-functional properties in order to eliminate false positive sensor data in the network. A data collection application may require data aggregation as its non-functional property in order to reduce traffic volume and expand the network's lifetime. Without breaking the generic architecture of TinyDDS, its 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. TinyDDS is the first middleware for WSN applications to flexibly configure the two types of non-functional properties.

This chapter describes the design, implementation and performance of TinyDDS. It discusses the layered architecture of TinyDDS, followed by details of each layer, application development process with nesC and Java, and a pluggable framework for non-functional properties. This chapter also evaluates TinyDDS' performance through blackbox and whitebox measurements in simulation and empirical experiments. TinyDDS is lightweight and efficient, and simplifies the development of publish/subscribe WSN applications.

BACKGROUND

This section overviews the publish/subscribe communication scheme and describes the standard DDS specification.

Publish/Subscribe Communication in WSNs

The publish/subscribe (pub/sub) communication scheme (Banavar et al., 1999; Eugster et al., 2003) is expected to significantly aid in developing and maintaining WSN applications by decoupling space and time among event source nodes (publishers) and sink nodes (subscribers) (Hadim & Mohamed, 2006; Wang et al., 2008; Henricksen & Robinson, 2006). In the pub/sub scheme, a subscriber has the ability to express its interest in an event or a pattern of events in order to be notified subsequently. Each interest is subscribed to a publisher(s), and the publisher(s) notifies an event to a subscriber(s) when the event matches a subscribed interest. Publishers do not need to know the number and locations of subscribers, and vice versa. Thus, publishers indirectly publish events to subscribers, and subscribers indirectly subscribe their interests to publishers. Moreover, publishers do not need to know the availability of subscribers, and vice versa. For example, subscribers may be active, sleeping or dead due to a lack of battery when a publisher publishes an event to them. Event subscription and publication are asynchronously transmitted among publishers and subscribers. By decoupling publishers and subscribers in space and time, the pub/sub scheme can improve scalability in terms of network size and traffic volume.

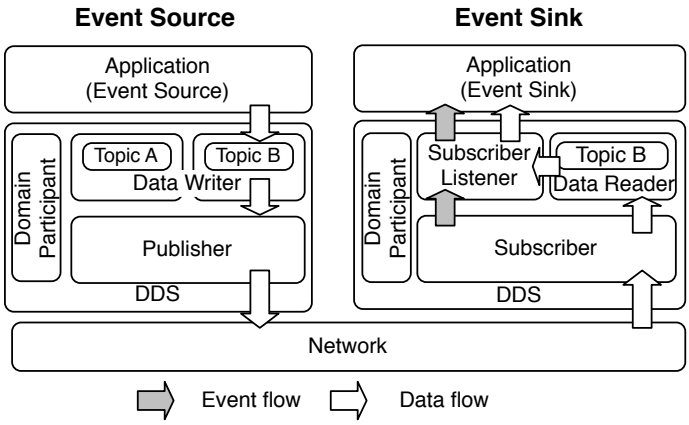
OMG DDS Specification

DSS is a specification that OMG standardizes for pub/sub middleware. It provides standard interfaces for event subscription and publication in Interface Definition Language (IDL). TinyDDS implements them with nesC and Java for the TinyOS and SunSPOT platforms, respectively. DDS consists of two layers: a lower-level fundamental layer called Data-Centric Publish-Subscribe (DCPS) and a higher-level optional layer called Data Local Reconstruction Layer (DLRL). DCPS defines a set of interfaces for event subscription and publication. Using the interfaces, each event is defined with an associated *topic*. The interfaces also allow applications to declare their intents to become publishers and subscribers and transmit event subscriptions/publications between publishers and subscribers. DLRL automatically

obtains events from a remote publisher and allows a subscriber to access the events as if they were locally available. Currently, TinyDDS implements DCPS only to minimize its memory footprint and processing overhead.

Figure 1 and 2 show key components in DDS. Figures 3 and 4 illustrate how these DDS components are used in the subscription and publication processes, respectively. Each node operates a single instance of `DomainPartipant` for each domain. A domain is a context to which a DDS application is associated. A `DomainPartipant` maintains references to all objects associated to the same domain.

Figure 1 DDS Architecture



When an event-sink application subscribes to an event(s), it instantiates `Subscriber` with the local `DomainParticipant` (Figure 3). Then, it creates an instance(s) of `Topic` according to the event(s) it is interested in. A topic uniquely identifies a particular event's content type or context. For each topic, the application instantiates `DataReader` and `SubscriberListener` as the access points for reading event data in the future (Figure 3). An event subscription is transmitted toward an event-source application(s) via `Subscriber`.

Similarly, an event-source application instantiates `Publisher` with the local `DomainParticipant` (Figure 4). It creates an instance(s) of `Topic` according to the event(s) it generates. For each topic, the application instantiates `DataWriter` as the access point for writing out event data in the future (Figure 4).

When an event-source application generates an event, it writes out the event to a `DataWriter`. Then, the event is transmitted toward an event-sink application(s) via `Publisher`. At a node where an event-sink application runs, a `Subscriber` monitors incoming event messages. If the application has subscribed to the topic of an incoming event, `Subscriber` informs the local `SubscriberListener` and `DataReader` that are associated with the event topic (Figure 1). Then, the `SubscriberListener` informs the event's arrival to the application, which in turn reads the event via `DataReader`.

Instead of receiving all incoming events of the subscribed topics, an event-sink application can filter them out with a `ContentFilteredTopic`, which derives `Topic` (Figure 2). `ContentFilteredTopic` is used to specify a subscription interest in the events whose contents satisfy certain criteria. For example, an event-sink application can specify an interest in the events whose

Figure 3 Standard DDS Interfaces

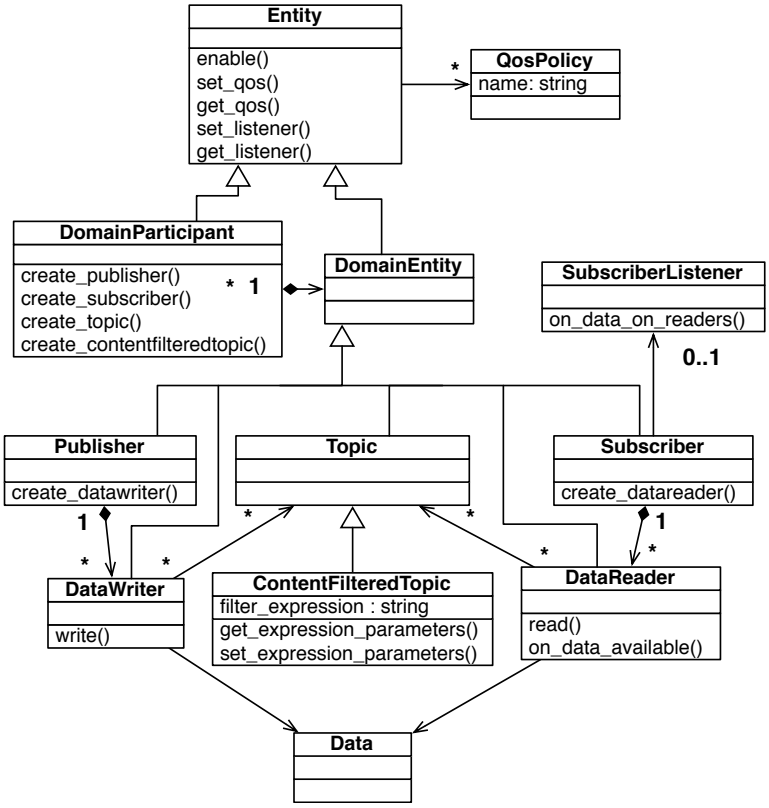
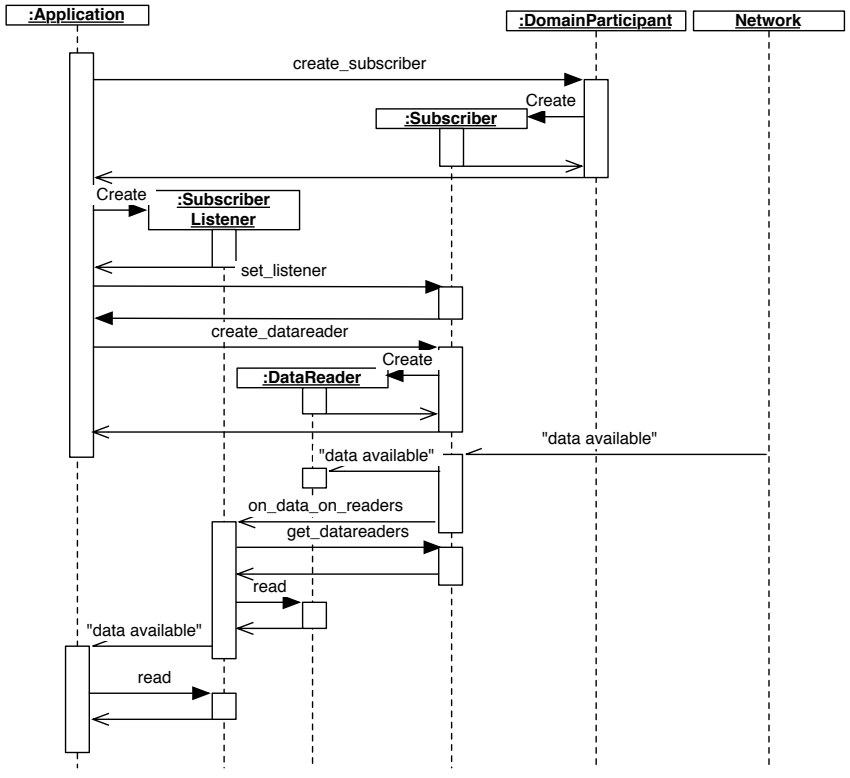


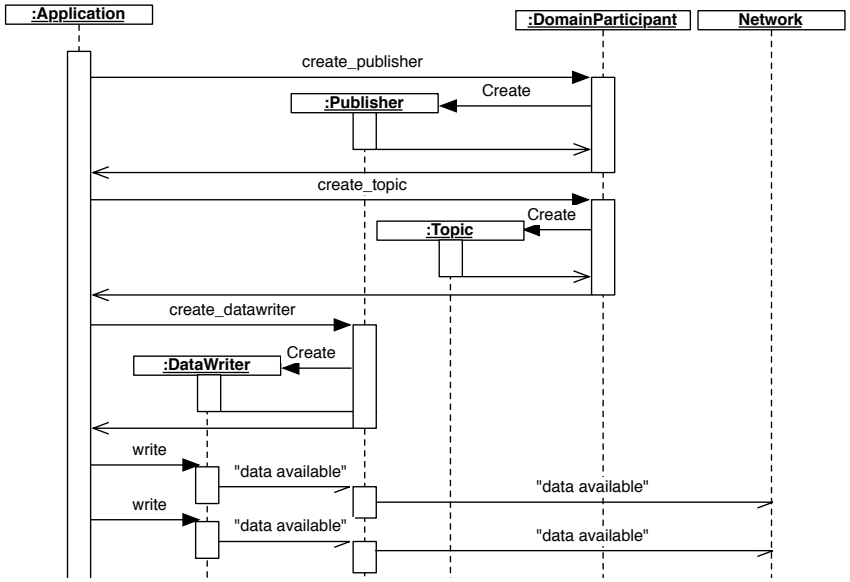
Figure 2 Subscription Process



topic is Temperature and whose contents is in between 100 and 150 degrees by defining "Temperature > 100 AND Temperature < 150" as criteria in a `ContentFilteredTopic`. In DDS, event filtering expression is described with a subset of SQL syntax.

Besides a set of standard DDS interfaces, the DDS specification defines no algorithms/protocols for event publication and subscription; they are left to DDS implementations. TinyDDS implements a subscription protocol and several publication protocols as subsequent sections discuss.

Figure 4 Publication Process



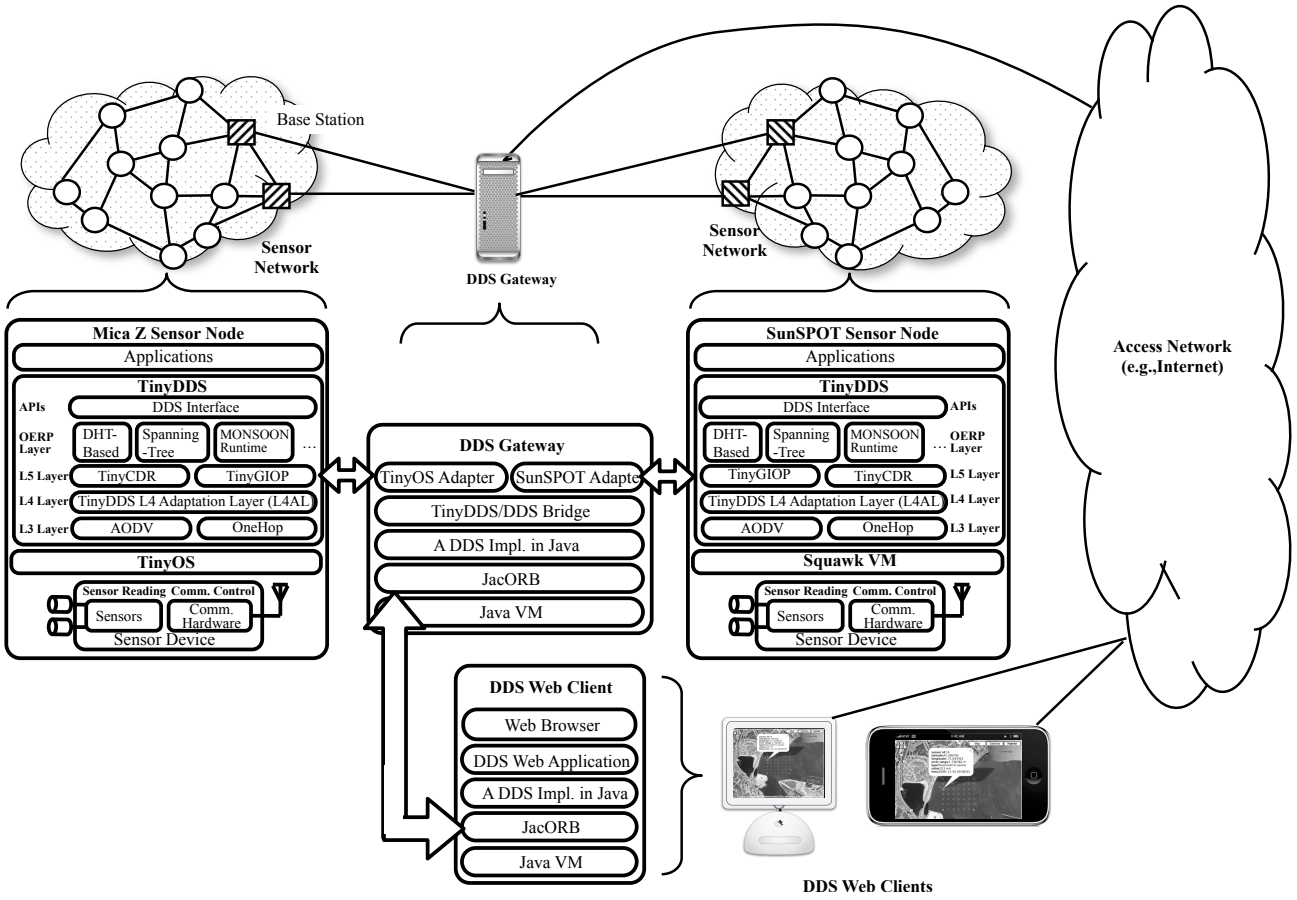
TINYDDS ARCHITECTURE

In Figure 5, TinyDDS architecture running in each sensor nodes is shown in the figure, with labeled TinyDDS. Currently, there are two implementations of TinyDDS, one for TinyOS platform, for example, Mica Z, Mica 2 or iMotes2 sensor nodes. The other is for Sun Microsystem's SunSPOT platform. The figure shows the architecture of TinyDDS for TinyOS running on TinyOS, i.e., inside a Mica Z mote, on the left hand side and the TinyDDS for SunSPOT platform on the right hand side. TinyDDS running on TinyOS-based sensor nodes is implemented in nesC programming language. On the other hand, TinyDDS running on SunSPOT platform is implemented on Java programming language and operates on top of Squawk Virtual Machine inside SunSPOT sensor nodes. With respect to TCP/IP reference model, TinyDDS operates in transport layer and work on top of any network layer (L3) implementation. TinyDDS follows Layer design pattern (Buschnmann, Meunier, Rohnert, Sommerland, & Stal, 1996) by separating different functionalities into different layers.

At the top layer, TinyDDS provides a subset of DDS interfaces to be used by applications. An application implemented on top of DDS can disseminate events to the network with associated topic and the events are captured by any subscribers, i.e., base station, who has interest on the topic of the events. The implementation of those interfaces, as described in the previous section, 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, spanning-tree 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 (Distributed Hash Table based) routing protocol. Moreover, OERP layer hides all event routing protocol implementation from developers. For example, if spanning-tree is used in OERP, the spanning-tree implementation wills forms the tree where the subscriber node is the root of the tree. The routing information, e.g., tree structure in spanning-tree, is performed when sensor nodes are started up and maintained automatically by the implementation in OERP layer. By using this OERP layer, TinyDDS frees developers from the need to manually maintain the event routing between nodes and the limitation of routing algorithm used in network layer, which generally depends on sensor node platform. The routing protocol in OERP layer utilizes low-level network layer implementation through a L5 layer called TinyGIOP. TinyGIOP encapsulates data into transportation messages and interacts with the DDS Gateway for exchanging data with DDS applications. Only the nodes, i.e., base station, that are physically connected to the DDS gateway through serial interface can exchange data with the DDS gateway. Beside, another component in this layer called TinyCDR provides an interchangeable data format, which allows different implementations of TinyDDS or DDS exchange data. For transmitting/receiving data to/from the other sensor nodes in the WSN, TinyGIOP utilizes a transport layer interface called TinyDDS L4 Adaption Layer (L4AL). L4AL allows TinyOS to operates with any network (L3) and MAC layer (L2) protocol, such as AODV and Zigbee (IEEE 802.15.4) respectively.

Figure 5 Architectural Components in TinyDDS



DDS Interfaces

In the top layer, TinyDDS provides an API for application developers. This API provides a subset of DDS for creating topics, subscribe to events of topics and publish events for particular topics. For each function, the implementation is provided so application developers do not need to implement that functionality themselves. The implementation for the DDS interfaces is written in nesC and Java programming language and optimized for small sensor nodes platform such as TinyOS and SunSPOT, respectively.

Listing 1 An Example TinyDDS Application with nesC

```
1  typedef struct {
2     cdr_short temperature;
3     cdr_long time;
4  } TempData_t;
5  Publisher_t publisher;
6  Topic_t topic;
7  DataWriter_t data_writer;
8  TempData_t temp_data;
9  command result_t StdControl.start() {
10    publisher = call DomainParticipant.create_publisher();
11    topic = call DomainParticipant.create_topic("TempSensor");
12    data_writer = call Publisher.create_datawriter(publisher , topic);
13    temp_data.temperature = TempSensor.read();
14    temp_data.time = call Time.getLow32();
15    call DataWriter.write(data_writer, serialize(data),
   sizeof(TempData_t));
16 }
```

Listing 2 An Example TinyDDS Application with Java

```
1  class TempData extends Data {
2     public short temperature;
3     public int time;
4  }
5  public class Application {
6     Publisher publisher;
7     Topic topic;
8     DataWriter dataWriter;
9     TempData tempData;
10    DomainParticipant domainParticipant;
11    public Application() {
12       domainParticipant = new DomainParticipant();
13       publisher = domainParticipant.create_publisher();
14       topic = domainParticipant.create_topic("TempSensor");
15       dataWriter = publisher.create_datawriter(topic);
16       tempData = new TempData();
17       tempData.temperature = TempSensor.read();
18       tempData.time = (new Date()).getTime();
19       dataWriter.write(data.marshall());
20    }
21 }
```

Listing 1 and Listing 2 show an example of an event source application implemented on top of TinyDDS using nesC and Java, respectively. In Listing 1, a user-defined data type is defined at line 1-4. Then, at line 10, a *Publisher* is created. Line 11-12, a *DataWriter* is created associate with topic "TempSemsor". At line 13 and 14, a sensor reading is captured from temperature sensor and also the current time is read from a local clock, both data are stored in a variable of the user-define data type. Finally, at line 15, the data in user-defined data type is serialized into byte stream and published through *DataWriter* interface. Listing 2 shows the same application implemented in Java. Because both applications are developed based on the same API, i.e., TinyDDS, the applications are very similar. Thus, application developer can easily port an application from one platform to the other platform.

Overlay Event Routing Protocols

The 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 that subscribes to the events. The published event is routed to each subscriber according to the event 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 property 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 spanning-tree 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 in the same platform.

Example of event routing protocols currently implemented for OERP layer are spanning-tree based, DHT-based and MONSOON. When spanning-tree based event routing protocol is used in OERP layer, subscriber nodes flood the subscription messages, i.e., the topic of interest, through their neighbors. The subscription messages have a counter that is increased hop-by-hop. Each node keeps the value of the counter of each incoming message, and its topic, and also the neighboring node who has lower counter value. Then, when a sensor node collect an event matched with the topic interested by a subscriber node, the event message is forwarded from the collecting node to a neighboring node who has lower counter of that topic. The event message is forwarded toward the subscriber node by climbing up the gradient, i.e., the counter value.

DHT-based event routing relies on unique information belongs to each sensor node such as sensor node ID in MicaZ or MAC address in SunSpot. Then, by using a hash function, a topic can be mapped to a sensor node ID or MAC address. This node can be called hashed node. Thus, when a sensor node subscribes to a topic, the subscription and the node information is sent to the hashed node that is associated with that topic. Similarly, when an event is published, the event is sent to the hashed node associated with the topic of the event. Consequently, the event is forwarded to subscribed node using subscribing information maintained by the hashed node.

In MONSOON, a biologically-inspired adaptation routing mechanism, event is delivered by *software agent*, from publishers to subscribers (Boonma & Suzuki, 2008a). In contrast with spanning-tree and DHT-based routing protocol, MONSOON employs a constrained-based evolutionary multiobjective optimization algorithm which allows agents to adapt to changed network condition and also self-healing against partial network failure.

TinyCDR

The Common Data Representation (CDR) (Object Management Group, 2007) is the format for exchanged data in DDS. CDR is the format for exchanging data in DDS standardized by OMG. CDR enables different parties, i.e., sensor nodes and client applications, which utilizes different programming languages, such as nesC or Java, to be able to exchange data. CDR defines standard data type with specific size and endian, which have to be followed by each party in order to guarantee seamlessly data exchanging. TinyCDR is a subset of CDR, which allows TinyDDS applications to directly exchange data with DDS applications and TinyDDS in different platform, such as between desktop application using DDS and sensor node application running in Mica Z and SunSPOT. Table 1 shows the mapping of primitive data type between CDR version 1.3, TinyCDR for nesC and for Java programming language. Listing 1 and Listing 2 show how the data type defined in TinyCDR for nesC and for Java, respectively, is used in application.

Table 1 CDR Mappings to nesC and Java

CDR Type	TinyCDR Type in nesC	TinyCDR Type in Java
char	cdr_char	byte
wchar	N/A	N/A
octet	cdr_octet	byte
short	cdr_short	short
unsigned short	cdr_ushort	int
long	cdr_long	int
unsigned long	cdr_ulong	long
long long	cdr_longlong	long
unsigned long long	cdr_ulonglong	N/A
float	cdr_float	float
double	cdr_double	double
long double	cdr_longdouble	double
boolean	cdr_boolean	boolean

In the table, TinyCDR does not support *wchar* type (wide character, i.e., Unicode characters) because it is not necessary in WSN environment. Also, TinyCDR for Java does not support *unsigned long long* type because there is no Java primitive type that has the same storage size for this data type. Beside primitive data types, TinyCDR also supports CDR constructed types such as *struct*, *union* and *array*. TinyCDR serializes constructed data structure into an octet stream, which is compatible with CDR octet stream. Therefore, TinyDDS applications can exchange data formatted in TinyCDR directly with DDS applications using CDR data format.

TinyGIOP

TinyGIOP defines message format use for exchanging between TinyDDS/DDS applications, based on General Inter-ORB Protocol (GIOP) version 1.3 (Object Management Group, 2007). GIOP is an abstract

protocol for communicating between object request brokers (ORBs). There are several concrete implementation based on GIOP such as Internet Inter-ORB Protocol (IIOP), an implementation of GIOP over TCP/IP, and HyperText Inter-ORB Protocol (HTIOP), an implementation of GIOP over HTTP. GIOP consists of three components; CDR, Interoperable Object Reference (IOR), and a set of message types. In TinyDDS, the CDR part of GIOP is addressed by TinyCDR while the message type is addressed by TinyGIOP. Given the limited resources of sensor nodes, IOR is not supported by TinyDDS.

TinyGIOP supports three message formats; *Request*, *Reply* and *CancelRequest*. When a TinyDDS application wants to communicate with the other TinyDDS application, for example, for subscribing to a topic, it sends out *Request* message. The message will be serialized and passed to lower level, i.e. OERP, or to DDS Gateway for delivering to DDS applications. *Reply* message is used for answering the request, e.g., when a TinyDDS application publish an event subscribed by another TinyDDS application, the publisher sends out *Reply* message to the subscriber. *CancelRequest* is used for withdraw request sending out earlier. Contrast with GIOP, TinyGIOP does not support object location message formats because there is no notion of object in TinyDDS.

TinyDDS L4 Adaptation Layer

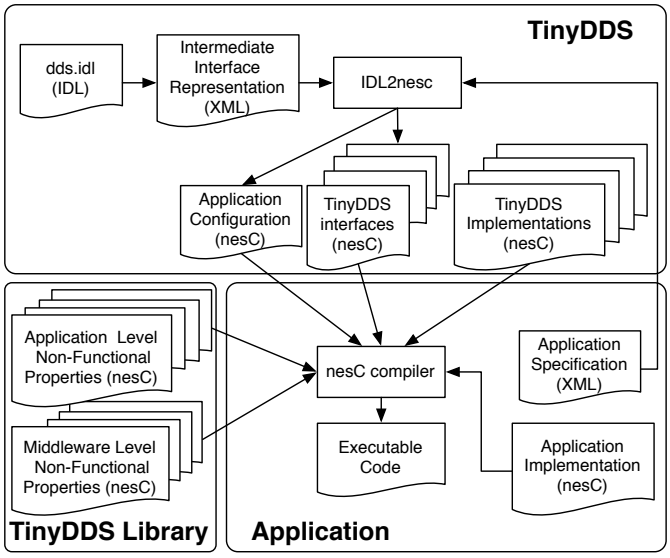
To access to low level physical network, TinyGIOP make use of low-level physical network through a network abstract layer called TinyDDS L4 Adaptation Layer (L4AL). This L4AL utilize Bridge design pattern to separate the real low-level physical network implementation from the higher-level overlay network. Thus, TinyDDS can be portable among different sensor node hardware, which utilize TinyOS, such as Mica Z , Mica 2 or iMotes2. 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 TinyGIOP and above OERP routing protocol 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.

Application Development with TinyDDS

Figure 6 shows the development model of a TinyDDS application on TinyOS platform. There are three main elements 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(Gamma, Helm, Johnson, & Vlissides, 1995) and describes how to connect each interfaces and implementation together. IDL2nesc also uses an *Application Specification*, written in XML, in order to generate appropriate Application Configuration. In particular, Application Configuration consults Application Specification how to connect comments together. Thus, developers can customize TinyDDS by adjusting the configuration in the Application Specification. 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. In particular, the event routing protocols

implemented for OERP layer needs to implement a specific nesC interface; therefore, they can be integrated with DDS implementation on the upper layer and L4AL component on the lower layer.

Figure 6 Application Development Model on TinyOS



The second element 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 provide 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 this functionality in the TinyDDS Library and the TinyDDS Library can be used in any application implemented on the same platform.

Listing 3 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_DataAggregrator" />
7        <component name="LedsC" />
8        <component name="SpanningTree" />
9     </includes>
10    <implementations>
11       <implementation component="SpanningTree" interface="OERP" />
12    </implementations>
13    <connections>
14       <connection from="Main.StdControl" to="*" />
15       <connection from="Application.DG"  to="DDS_DataAggragation" />
16       <connection from="Application.Leds" to="LedsC" />
17    </connections>
```

The third element is the application. Every application implemented on TinyDDS 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 performs a certain task such as data collection and event detection. 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 for TinyOS platform.

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

Figure 7 Application Development Model in SunSPOT

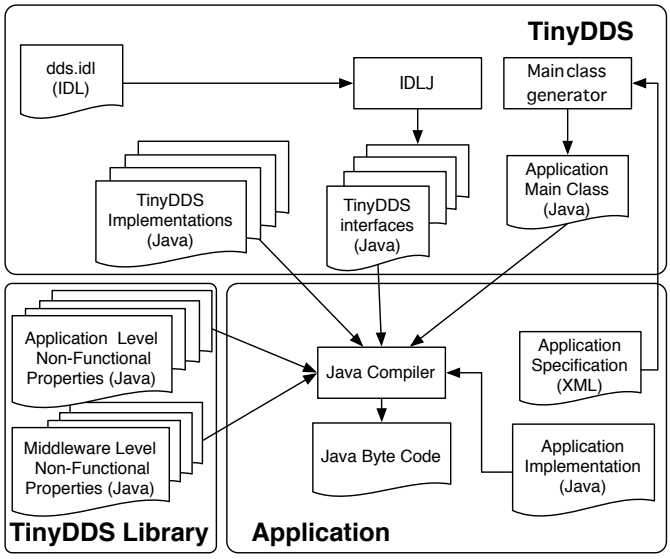


Figure 7 shows the application development model in SunSPOT. Similar to the application development model on TinyOS, there are three main elements, TinyDDS middleware, TinyDDS library and the application implementation. Contrast with the TinyOS model, Java is used as programming language instead of nesC. Also, a Java compiler, i.e., javac, is used for combining all parts into a Java byte code suitable for deploying in a SunSPOT sensor node. Also, Sun Microsystem's IDLJ is used for converting DDS's IDL to Java interfaces and the main class generator program is used for processing Application Specification and generating the main class used by Java compiler.

DDS Gateway

Figure 5 shows that TinyDDS uses TinyGIOP to communicate with the DDS gateway in order to exchange data with DDS applications. The DDS gateway is a Java application that interacts with TinyDDS running in sensor nodes through serial port using TinyOS' serial adapter and SunSPOT serial adapter (i.e., Host API) Java class. The DDS gateway uses JacORB (Brose, 1997) and a Java implementation of DDS (Allaoui, Yehdih, & Donsez, 2005) to communicate with another DDS application. A TinyDDS/DDS bridge operates on top of DDS implementation and communicates with TinyGIOP to exchange data between TinyDDS and DDS. When a DDS client subscribes to a topic, the

subscription information is distributed on the access network over DDS. As a consequence, DDS gateway can intercept the subscription information and store it in a subscription list. Thus, when a sensor node publishes an event, the event is distributed in the sensor network and intercepted by DDS gateway. Then, if the topic of the published event is matched with a topic on the subscription list. This filtering mechanism in DDS gateway is similar to traditional TCP/IP bridge operation. Moreover, by using DDS as the backend, the DDS gateway can be deployed in variable configurations, e.g., multiple sensor networks connect to a single DDS gateway or multiple DDS gateways connect to a sensor network. Each DDS gateway supports thread-per-connection, thread pooling, connection pooling and Reactor.

In particular, when a message is pushed from TinyGIOP in a sensor node to DDS gateway, the TinyDDS/DDS bridge translate message into DDS format, i.e., encapsulate with GIOP header, and send out to DDS network. This is called **downstream** message transmissions because the message is sent from source (sensor nodes) to sink (client applications). On the other hand, when the DDS gateway receives messages destine to the sensor network from the DDS network, TinyDDS/DDS bridge translates the message into TinyDDS format, i.e., encapsulate with TinyGIOP header, and injects the message into sensor nodes through serial interface. This is called **upstream** message transmission. Also, the message from one TinyDDS will be passed to another TinyDDS, for example, TinyDDS on a TinyOS nodes can pass message through its TinyGIOP to DDS Gateway, which will also pass the message to TinyDDS on the SunSPOT as well.

To manage the flow control from a WSN to an access network, TinyDDS utilizes the hop-by-hop flow control mechanisms available at the MAC layer of TinyOS and SunSPOT. In particular, the flow control mechanisms at the MAC layer allow a sensor node to send out a packet only when it receives acknowledgment from the destination node; there fore, the source node cannot overwhelm the destination node. However, TinyDDS does not consider end-to-end flow control. For the flow control from an access network to a WSN, TinyDDS uses TCP's end-to-end flow control.

DDS Web Clients

On the lower part of the Figure 5, a DDS web client is shown connected to DDS gateway. Currently, a DDS web client is implemented as a Java application provides HTTP service to any web browser. The DDS web client is able to operate on ordinary desktop computers or mobile devices such as Apple's iPhone. By using JacORB, the DDS web client can communicate with DDS gateway in order to subscribe to data published from sensor networks and show the result on the Google map. Figure 8 and Figure 9 show examples of web interface running on a desktop computer and an iPhone respectively. In the Figures, the small dots show location of each sensor node, bright dots represent sensor nodes, which report data.

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 and Figure 7. The application-level non-functional properties accelerate 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 data and event. Moreover, utilizing non-functional properties can reduce the application

Figure 8 A DDS Web Client on a Desktop Computer

Figure 9 A DDS Web Client on an iPhone

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.

Application-Level Non-Functional Properties

In the application level, non-functional properties in TinyDDS help 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 pluggable, application developers can include this in their application using Application Specification (see section *Application Development with TinyDDS*). In addition, library developers can develop any non-functional components, such as network security and persistence storage, which are reusable and pluggable to all TinyDDS applications.

Listing 4 An Example of nesC Application Using Data Aggregation

```
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(ts_topic,
   AVERAGE, listener);
11 }
12 event ReturnCode_t SubscriberListener.data_available(Topic_t topic) {
13    Data data;
14    if (topic == ts_topic) {
15       data = call DataAggregator.read(data_aggregrator);
16       // processing aggregated data..
17    }
18 }
```

Listing 4 shows a fragment of an application using data aggregation in nesC. Instead of using `DataReader` for collecting data (see Figure 2), this application uses `DataAggregrator`. 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. Listing 5 shows the same application implemented in Java.

From the listing, Java implementation of TinyDDS adopts Observer Design Pattern. So, when an event is collected, SubscriberListener informs the application by calling eventNotified. Then, the application can collect aggregated data through an instance of DataAggragation class.

Listing 5 An Example of Java Application Using Data Aggregation

```
1  public class Application implement Observer {
2     Subscriber subscriber;
3     Topic topic;
4     DataAggregator dataAggregrator;
5     SubscriberListener listener;
6     DomainParticipant domainParticipant;
7     public Application() {
8        domainParticipant = new DomainParticipant();
9        subscriber = domainParticipant.create_ subscriber();
10       topic = domainParticipant.create_topic("TempSensor");
11       listener  = new SubscriberListener(topic, this);
12       subscriber.set_listener(listener);
13       data_aggregrator = subscriber.create_data_aggregrator(topic,
   AVERAGE, listener);
14    }
15    public void eventNotified(Object orig, Object event) {
16       TempData data;
17       if(orig == listener) {
18          data = data_aggregrator.read();
19       }
20    }
21 }
```

Middleware-Level Non-Functional Properties

TinyDDS supports three non-functional properties in the middleware level, the pluggable routing protocols in OERP layers, concurrency, and the QoS policy. Application developers can specify the concurrency model used in TinyDDS. In particular, TinyDDS supports three concurrency model, thread-per-event, thread-per-event-topic and Reactor (Pyarali, Harrison, Schmidt, & Jordan, 1997). In the thread-per-event model, TinyDDS creates a thread, e.g., a Task in TinyOS or a thread in Java, 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. 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 Reactor model, it is mapped to single-threaded mode in TinyOS and SunSPOT.

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 and Listing 7 show fragment of an application using latency budget QoS policy in nesC and Java, respectively. 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 6 Latency Budget QoS Parameters Settings in nesC.

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

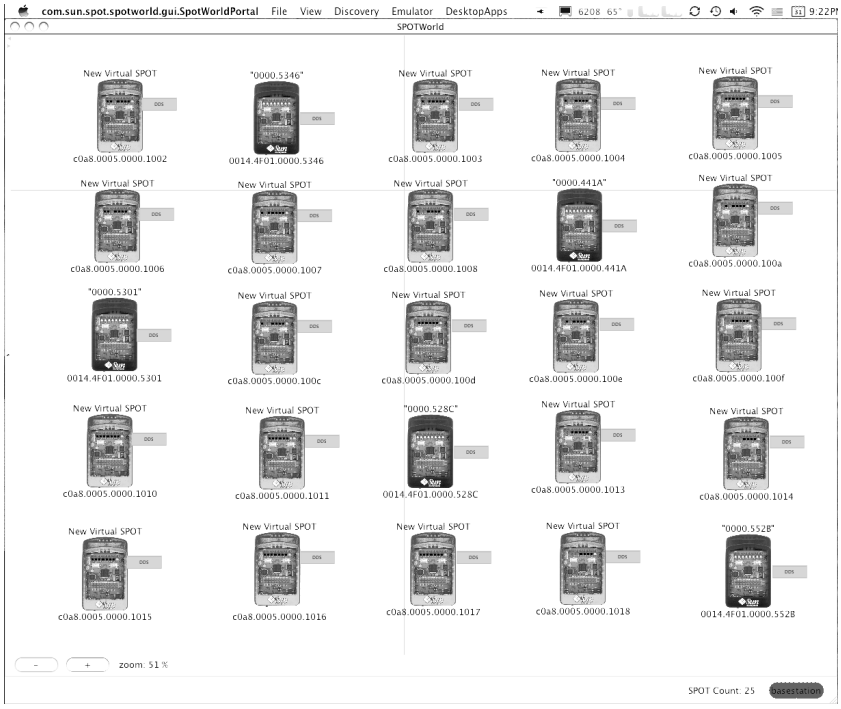
Listing 7 Latency Budget QoS Parameters Settings in Java.

```
1  public class Application {
2     Publisher publisher;
3     Topic topic;
4     DataWriter dataWriter;
5     TempData tempData;
6     DomainParticipant domainParticipant;
7     QosPolicy qos;
8     public Application() {
9        domainParticipant = new DomainParticipant();
10       publisher = domainParticipant.create_publisher();
11       qos = QosPolicy.create_latency_budget_qos(100);
12       topic = domainParticipant.create_topic("TempSensor");
13       topic.set_qos(qos);
14       dataWriter = publisher.create_datawriter(topic);
15       tempData = new TempData();
16       tempData.temperature = TempSensor.read();
17       tempData.time = (new Date()).getTime();
18       dataWriter.write(data.marshall());
19    }
20 }
```

EVALUATION

This section evaluates TinyDDS through simulations and empirical experiments. In all simulations and experiments, each node runs an application that accepts a one-time event subscription from a subscriber and publishes subsequent events to the subscriber. It is configured with an application specification shown in Listing 3. The duration of each simulation/experiment is 120 seconds, and 25 nodes transmit an event to the base station (i.e., subscriber) every 2 seconds. The TinyOS-based implementation of TinyDDS is evaluated with 25 nodes simulated on the PowerTOSSIM simulator with the MICA 2 power consumption model (Shnayder, Hempstead, Chen, Allen, & Welsh, 2004). The SunSPOT-based implementation of TinyDDS is evaluated with 20 nodes simulated on the Solarium emulator (Goldman, 2008) and 5 nodes deployed on real/physical SunSPOT platforms. Figure 10 shows a screenshot of the Solarium emulator. 20 light gray SunSPOT icons represent simulated nodes, and 5 dark gray SunSPOT icons represent real nodes.

Figure 10 A Screenshot of the Solarium Emulator showing 25 simulated and real nodes.



On TinyOS, TinyDDS is compared with Surge, which is a simple data collection application bundled in TinyOS. For a purpose of performance comparison, TinyDDS uses a spanning tree-based protocol as its OERP as Surge does. On SunSPOT, a Surge-like data collection application is implemented to for the comparison with TinyDDS. (It uses a spanning tree-based protocol too.)

Per-Packet Header Overhead

In order to evaluate how much data TinyDDS requires to transmit an event, Table 2 shows the header overhead at each layer of TinyDDS. Packet size is limited at 48 bytes on TinyOS. Given this limit, TinyDDS can transmit event data of 16 bytes per packet; packet overhead is 32 bytes (66%) per packet. Given 16 bytes, TinyDDS can transmit a sequence/array of two unsigned long long values. (Unsigned long long is the largest fixed-length IDL type; it occupies 8 bytes.) Also, It can contain 16 of the shortest fixed-length IDL data such as char, octet and Boolean per packet. Surge's packet overhead is approximately 11 bytes (40%) per packet. Packet size is limited at 64 bytes on SunSPOT; packet overhead is 52 bytes (82%) per packet. TinyDDS can transmit event data of 12 bytes per packet. In a Serge-like data collection application, packet overhead is 32 bytes (66%) per packet.

TinyDDS incurs larger header overhead because it encapsulates more information in each packet; for example, a timestamp from DDS, routing information from OERP and session information from TinyGIOP. With these extra information, TinyDDS can perform advanced functionalities that simple data collection applications do not have. For example, TinyDDS can use timestamp information to calculate the delay in an event publication and prioritize packets hop by hop. Given this consideration, the authors of the chapter believe that the measured header overhead is acceptable and small enough for a number of WSN applications.

Table 2 Per-Packet Header Overhead of TinyDDS on TinyOS and SunSPOT

Size (Bytes)	**TinyOS**	**SunSPOT**
Event Data	16	12
DDS Overhead	10	21
OERP (Spanning Tree) Overhead	4	4
L5 Overhead	8	12
L4 Overhead	6	8
L3 (OneHop) Overhead	4	8
Total Overhead	32	52
Total Size	48	64

Memory Footprint

Table 3 shows the memory footprint of TinyDDS on TinyOS and SunSPOT. Without running applications on TinyOS, TinyDDS consumes 36,132 bytes of Flash memory and 3,360 bytes of RAM. Including an application, memory footprint slightly increases to 36,246 bytes in Flash memory and 3,392 bytes of RAM. Surge consumes 34,430 byte of Flash memory and 1,929 byte of RAM. The difference of memory footprint between TinyDDS and Surge is very small on TinyOS; less than 2 KB on both Flash memory and RAM.

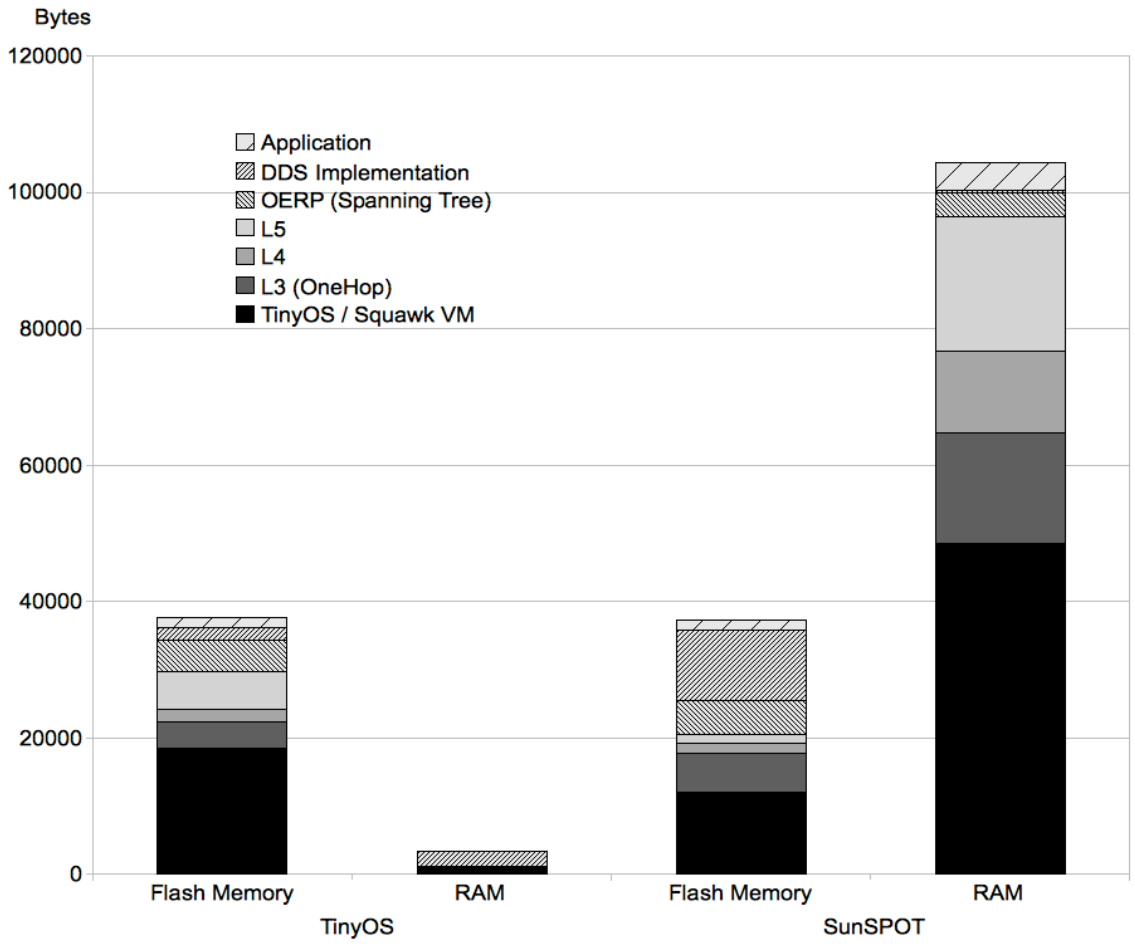
Without running applications on SunSPOT, TinyDDS consumes 35,832 bytes of Flash memory and 100,268 bytes of RAM. RAM consumption is higher on SunSPOT than TinyOS because SunSPOT requires to copy every executable code, including the Squawk VM and applications, and execute them in

RAM. Including an application, TinyDDS consumes 37,285 bytes of Flash memory and 104,404 bytes of RAM. A Surge-like data collection application consumes about 29,519 bytes of Flash memory and 82,739 bytes of RAM. The difference of memory footprint between TinyDDS and a Surge-like application is small enough on SunSPOT. TinyDDS is implemented lightweight, and it can operate in resource-limited nodes such as MICA2. Figure 11 shows the breakdown of memory footprint in each layer in TinyDDS.

Table 3 Memory Footprint of TinyDDS on TinyOS and SunSPOT.

Memory Footprint (Bytes)	**TinyOS**		**SunSPOT**	
	Flash memory	**RAM**	**Flash memory**	**RAM**
Application	1440	34	1453	4136
DDS Implementation	1784	2240	10388	404
OERP (Spanning Tree)	4644	172	4907	3364
L5	5452	175	1352	19800
L4	1827	150	1457	11892
L3 (OneHop)	3905	205	5693	16248
TinyOS/ Squawk VM	18520	418	12035	48560

Figure 11 Memory Footprint of Components on TinyOS and SunSPOT Platform.



Processing and Power Efficiency

When TinyDDS is deployed on TinyOS, the average total latency is 0.79 second to route an event between source and destination nodes in a single hop. Of this total latency, TinyDDS spends 0.08 second; 0.03 second on a source node and 0.05 second on a destination node. This means TinyDDS occupies approximately 10% of the total latency.

When TinyDDS is deployed on SunSPOT, the average total latency is 0.675 second to route an event between source and destination nodes in a single hop. Of this total latency, TinyDDS spends 0.0247 second; 0.0128 second on a source node and 0.0119 second on a destination node. Table 4 shows the breakdown of the latency on both source and destination nodes. TinyDDS' latency occupies approximately 10% of the total latency. When a Serge-like application is used for routing event, the average total latency is 0.58 second to route an event in a single hop. Thus, latency of event routing in TinyDDS is about 10% higher than in Surge. This difference is small enough.

Table 5 shows the average and standard deviation (SD) of power consumption from 25 simulated Mica 2 nodes using TinyDDS on TinyOS platform and from five actual SunSPOT nodes and 20 emulated nodes running TinyDDS. Without a subscriber, TinyDDS transmits no data; thus, its power consumption remains small. In contrast, Surge always transmits data to the base station; it consumes much more power than TinyDDS. With a subscriber, TinyDDS consumes a comparable amount of power compared with Surge. In SunSPOT, without a subscriber, TinyDDS transmits no data; thus, its power consumption remains small. In contrast, a simple data collection application always transmits data to the base station; it consumes much more power than TinyDDS. With a subscriber, TinyDDS consumes a comparable amount of power compared with the simple data collection application. TinyDDS is implemented power efficient.

Table 5 Power Consumption in Mica 2 (TinyOS) and SunSPOT

Power Consumption (mW)		**Mica 2 (TinyOS)**		**SunSPOT**	
		TinyDDS	**Surge**	**TinyDDS**	**A Data Coll. App.**
Without a Subscriber	Average	189.59	3743.0	295.2	5544.4
	SD	61.24	76.03	89.5	76.03
With a Subscriber	Average	3900.9	3924.97	5323.75	5601.3
	SD	52.55	76.03	83.4	75.4

Table 4 Latency of each layer on SunSPOT.

	Source Node (msec)	**Destination Node (msec)**
DDS	0.1	0.3
OERP (Spanning-Tree)	0.5	0.75
L5	5.5	10.15
L4	4.3	0.3
L3	2.4	0.4
Total	12.8	11.9

Size of Application Source Code

With the TinyOS-based implementation of TinyDDS, only 60 lines of nesC code are required to implement the same application as Surge. Surge is implemented with 300 lines of nesC code. Moreover, it takes less than 3 seconds for idl2nesc to nesC source code. With the SunSPOT-based implementation of TinyDDS, 80 lines of Java code are required for implementing a Surge-like application. On the other hand, without TinyDDS, more than 350 lines of Java code are required to implement the same application from scratch. It takes less than 10 seconds for idl2j to generate Java code. These results demonstrate that TinyDDS effectively simplifies the development of WSN applications.

RELATED WORK

This chapter describes a set of extensions to the authors' prior work (Boonma & Suzuki, 2008b; Boonma & Suzuki, 2009b). While the prior work studied TinyDDS on TinyOS, this chapter investigates it on both TinyOS and SunSPOT. Moreover, this chapter reveals TinyDDS' performance implications in more detail. An event routing protocol of TinyDDS, called MONSOON, is investigated in (Boonma & Suzuki, 2008a; Boonma & Suzuki, 2009a), while it is out of scope of this chapter.

There exist several pub/sub middleware for WSNs. TinyCubes, Mires and Runes are similar to TinyDDS in that they implement pub/sub communication on TinyOS and provide reconfigurable middleware services to customize application-level non-functional properties (Marrón et al., 2005; Souto, et al., 2005; Costa et al., 2007; Nam et al., 2008). However, unlike TinyDDS, they do not consider middleware-level non-functional properties and interoperability between WSNs and access networks. SMC is pub/sub middleware for body-area sensor networks (Keoh, et al., 2007). It is assumed to operate on a Java VM atop powerful Linux node; memory footprint and power efficiency are not important issues in SMC. In contrast, TinyDDS assumes resource-limited nodes as its target platforms. SMC does not consider middleware-level non-functional properties, programming language interoperability and protocol interoperability as TinyDDS does. DSWare is pub/sub middleware that makes middleware-level non-functional properties (e.g., data storage and caching policies) configurable for WSN applications (Li et al., 2004). However, it does not consider application-level non-functional properties, programming language interoperability and protocol interoperability as TinyDDS does.

Schönherr et al. (2008) propose a clustered event routing protocol for WSNs. It allows nodes to form clusters and transmit event data to destinations (subscribers) via cluster head nodes. Costa and Picco (2005) propose a semi-probabilistic routing protocol in which event data are randomly broadcasted when intermediate nodes do not know subscribers. TinyDDS does not focus on particular event routing protocols. Instead, it focuses on its generic and pluggable framework to support a wide range of event routing protocols.

Li et al. (2004), Yoneki and Bacon (2005) and Sivaharan et al. (2005) propose event subscription languages for the pub/sub scheme in WSNs. Yoneki and Bacon (2005) extend a traditional subscription language by introducing the expressiveness for spatial and temporal concerns. Li et al. (2004) investigates an SQL-like subscription language that supports the notion of confidence in subscription and event correlation. Sivaharan et al. (2005) propose an extensible subscription language, called Filter Expression Language (FEL), to define topic, content and context filtering policies for published events. In contrast, TinyDDS currently does not focus on subscription semantics but reuse the standard subscription language in the DDS specification.

Several research efforts have focused on the interoperability between WSNs and access networks. Girod et al. (2004), Pietzuch et al. (2004), Spiess et al. (2006) and Hunkeler et al. (2008) propose

interoperable middleware/frameworks. Adam et al. (2004), Shu et al. (2006) and Schott et al. (2007) investigate TCP/IP-based and other protocol stacks. These work provide interoperability by unifiying low-level (L2 to L4) protocols between WSNs and access networks. In contrast, TinyDDS provides interoperability by introducing an interoperable session (L5) protocol over heterogeneous L2 to L4 protocols. Given the fact that heterogenity have been increasing in WSN platforms, TinyDDS retains the heterogenity at L2 to L4 rather than unifying them. Moreover, all of these existing work do not provide programming language interoperability. They also do not consider the pub/sub communication scheme as a networking/programming abstraction.

CONCLUSION

TinyDDS is pub/sub middleware that allows applications to interoperate regardless of programming languages and protocols across the boundary of WSNs and access networks. Moreover, it allows WSN applications to have fine-grained control over application-level and middleware-level non-functional properties and flexibly specialize in their own requirements. Evaluation results demonstrate that TinyDDS is lightweight and efficient. They also show that TinyDDS simplifies the development of publish/subscribe WSN applications.

Several future extensions are planned for TinyDDS. One of them is to investigate an end-to-end flow control mechanism between WSNs and access networks. Another is to evaluate various I/O and resource management mechanisms in scalability measurements with varying network sizes and traffic patterns. Particularly, the authors of the chapter are interested in studying staged event-driven architecture (SEDA) (Welsh, Culler, & Brewer, 2001) on a gateway node and the Proactor design pattern (Pyarali, Harrison, Schmidt, & Jordan, 1997) on individual sensor nodes.

REFERENCES

Adam, D., Thiemo, V., & Juan, A. (2004). Making TCP/IP Viable for Wireless Sensor Networks. *European Workshop on Wireless Sensor Networks* (p. 4). Berlin, Germany: Springer.

Allaoui, F., Yehdih, A., & Donsez, D. (2005, August 20). *Open-source Java-based DDS (Data Distribution Service) Implementation*. Retrieved August 25, 2008, from Open-source Java-based OMG DDS Implementation: http://www-adele.imag.fr/users/Didier.Donsez/dev/dds/readme.html

Banavar, G., Candra, T. D., Strom, R. E., & Sturman, D. C. (1999). A Case for Message Oriented Middleware. *International Symposium on Distributed Computing* (pp. 1-18). Bratislava, Slovak Republic: Springer.

Boonma, P., & Suzuki, J. (2008). Exploring Self-star Properties in Cognitive Sensor Networking. *International Symposium on Performance Evaluation of Computer and Telecommunication Systems.* Edinburgh, Scottland: IEEE/SCS.

Boonma, P., & Suzuki, J. (2008). Middleware Support for Pluggable Non-Functional Properties in Wireless Sensor Networks. *International Workshop on Methodologies for Non-functional Properties in Services Computing* (pp. 360-367). Honolulu, Hawaii: IEEE.

Boonma, P., & Suzuki, J. (2009). Self-Configurable Publish/Subscribe Middleware for Wireless Sensor Networks. *International Workshop on Personalized Networks.* Las Vegas, Nevada: IEEE.

Boonma, P., & Suzuki, J. (2009). Toward Interoperable Publish/Subscribe Communication between Wireless Sensor Networks and Access Networks. *International Workshop on Information Retrieval in Sensor Networks*. Las Vegas, Nevada: IEEE.

Brose, G. (1997). JacORB: Implementation and Design of a Java ORB. *International Working Conference on Distributed Aplications and Interoperable Systems* (pp. 143-154). Cottbus, Germany: Chapman & Hall.

Buschnmann, F., Meunier, R., Rohnert, H., Sommerland, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture - A: System of Patterns*. Wiley and Sons.

Costa, P., & Picco, G. P. (2005). Publish-Subscribe on Sensor Networks: A Semi-Probabilistic Approach. *International Conference on Mobile Adhoc and Sensor Systems* (p. 332). Washington DC: IEEE.

Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G. P., & Zachariadis, S. (2007). A Reconfigurable Component-based Middleware for Networked Embedded Systems. *Springer Journal of Wireless Information Networks , 14* (2), 149-162.

Estrin, D., Govindan, R., Heidemann, J., & Kumar, S. (1999). Next Century Challenges: Scalable Coordination in Sensor Networks. *International Conference on Mobile Computing and Networks* (pp. 263-270). Seattle, Washington: ACM.

Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A.-M. (2003). The Many Faces of Publish/Subscribe. *ACM Computing Surveys , 35* (2), 114-131.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Gay, D., Levis, P., Behren, R. v., Welsh, M., Brewer, E., & Culler, D. (2003). The nesC Language: A Holistic Approach to Networked Embedded Systems. *Programming Language Design and Implementation* (pp. 1-11). San Diego, California: ACM.

Girod, L., Elson, J., Cerpa, A., Stathopoulos, T., Ramanathan, N., & Estrin, D. (2004). Emstar: A Software Environment for Developing and Deploying Wireless Sensor Network. *USENIX Technical Conference* (pp. 24-38). Boston, Massachusetts: USENIX.

Goldman, R. (2008, June 1). *Using the SPOT Emulator in Solarium*. Retrieved December 24, 2008, from SunSPOTWorld: http://www.sunspotworld.com/docs/Blue/SunSPOT-Emulator.pdf

Hadim, S., & Mohamed, N. (2006). Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online , 7* (3), 1.

Hadim, S., & Mohamed, N. (2006). Middleware for Wireless Sensor Networks: A Survey. *International Conference on Communication System Software and Middleware*, (pp. 1-7). New Delhi, India.

Henricksen, K., & Robinson, R. (2006). A Survey of Middleware for Sensor Networks: State-of-the-art and Future Directions. *International Workshop on Middleware for Sensor Networks* (pp. 60-65). Melbourne, Australia: ACM.

Hunkeler, U., Truong, H. L., & Stanford-Clark, A. (2008). MQTT-S — A Publish/Subscribe Protocol for Wireless Sensor Networks. *Communication Systems Software and Middleware and Workshops* (pp. 791-798). Dublin, Ireland: ICST.

Keoh, S. L., Dulay, N., Lupu, E., Twidle, K., Schaeffer-Filho, A. E., Sloman, M., et al. (2007). Self-Managed Cell: A Middleware for Managing Body-Sensor Networks. *International Conference on Mobile and Ubiquitous Systems* (pp. 1-5). Philadelphia, Pennsylvania: IEEE.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., et al. (2005). TinyOS: An Operating System for Sensor Networks. In W. Weber, J. Rabaey, & E. Aarts, *Ambient Intelligence* (pp. 115-148). Berlin, Germany: Springer.

Li, S., Lin, Y., Son, S. H., Stankovic, J. A., & Wei, Y. (2004). Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. *Springer Telecommunication Systems , 26* (2-4), 351-368.

Marchiori, A., & Han, Q. (2008). A Foundation for Interoperable Sensor Networks with Internet Bridging. *Workshop on Embedded Networked Sensor.* Charlottesville, Virginia: ACM.

Marrón, P. J., Lachenmann, A., Minder, D., Gauger, M., Saukh, O., & Rothermel, K. (2005). Management and Configuration Issues for Sensor Networks. *Wiley International Journal of Network Management , 15* (4), 235-253.

Nam, C.-S., Jeong, H.-J., & Shin, D.-R. (2008). Design and Implementation of the Publish/Subscribe Middleware for Wireless Sensor Networks. *International Conference Networked Computing and Advanced Information Management* (pp. 270-273). Gyeongju, South Korea: IEEE.

Object Management Group. (2007). Common Object Request Broker Architecture (CORBA) Specification, Version 3.1; Part 2: CORBA Interoperability. Retrieved August 25, 2008

Object Management Group. (2007). Data Distribution Service (DDS) for real-time systems, v1.2. Retrieved August 25, 2008

Pietzuch, P., Ledlie, J., Shneidman, P., Roussopoulos, M., Welsh, M., & Seltzer, M. (2004). Network-Aware Operator Placement for Stream-Processing Systems. *International Conference on Data Engineering* (p. 49). Atlanta, Georgia: IEEE.

Pyarali, I., Harrison, T., Schmidt, D. C., & Jordan, T. D. (1997). Proactor -- An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. *Pattern Languages of Programming Conference.* Monticello, Illinois: Washington University.

Romer, K., Kasten, O., & Mattern, F. (2002). Middleware Challenges for Wireless Sensor Networks. *ACM Mobile Computing and Communications Review , 6* (4), 59-61.

Schönherr, J. H., Parzyjegla, H., & Mühl, G. (2008). Clustered Publish/Subscribe in Wireless Actuator and Sensor Networks. *International Workshop on Middleware for Pervasive and Ad-Hoc Computing* (pp. 60-65). Leuven, Belgium: ACM.

Schott, W., Gluhak, A., Presser, M., Hunkeler, U., & Tafazolli, R. (2007). e-SENSE Protocol Stack Architecture for Wireless Sensor Networks. *Mobile and Wireless Communications Summit* (pp. 1-5). Budapest, Hungary: IST.

Shnayder, V., Hempstead, M., Chen, B.-r., Allen, G., & Welsh, M. (2004). Simulating the Power Consumption of Large-Scale Sensor Network Applications. *International Conference on Embedded Networked Sensor Systems* (pp. 188-200). Baltimore, Maryland: ACM.

Shu, L., Wang, J., Xu, H., Jinsung, C., & Sungyoung, L. (2006). Connecting Sensor Networks with TCP/IP Network. *International Workshop on Sensor Networks* (pp. 330-334). Harbin, China: Springer.

Simon, D., & Cifuentes, C. (2005). The Squawk Virtual Machine: Java™ on the Bare Metal. *Conference on Object Oriented Programming Systems Languages and Applications* (pp. 150-151). San Diego, California: ACM.

Sivaharan, T., Blair, G., & Coulson, G. (2005). GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing. *On the Move to Meaningful Internet Systems* (pp. 732-749). Agia Napa, Cyprus: Springer.

Souto, E., Guimarães, G., Vasconcelos, G., Vieira, M., Nelson, R., Ferraz, C., et al. (2005). Mires: A Publish/Subscribe Middleware for Sensor Networks. *Springer Personal Ubiquitous Computing , 10* (1), 37-44.

Spiess, P., Vogt, H., & Jütting, J. (2006). Integrating Sensor Networks With Business Processes. *Real-World Sensor Networks Workshop.* Uppsala, Sweden: ACM.

Wang, M.-M., Cao, J.-N., Li, J., & Dasi, S. K. (2008). Middleware for Wireless Sensor Networks: A Survey. *Springer Journal of Computer Science , 23* (3), 305-326.

Welsh, M., Culler, D., & Brewer, E. (2001). SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *Symposium on Operating Systems Principles* (pp. 230-243). Banff, Canada: ACM.

Yoneki, E., & Bacon, J. (2005). Unified Semantics for Event Correlation over Time and Space in Hybrid Network Environments. *International Conference on Cooperative Information Systems* (pp. 366-384). Agia Napa, Cyprus: IFIP.