Chapter 7: Middleware Design in Wireless Sensor Networks

This chapter will introduce the middleware architecture of a WSN. Our discussions are based on the summarization of [Miaomiao08]. If the readers want to understand more details, please refer to [Miaomiao08] for a comprehensive survey.

7.1 Introduction

Typically, the network protocol stack can be classified into 5 layers (from top to bottom): Application layer, Transport layer, Routing layer, MAC layer, Physical layer. As an example, Crossbow Inc. motes (i.e., WSN nodes) allow a user to use NesC (similar to C language) to build sensor network control programs. As shown in Figure 1, a user builds those programs in Application layer in order to control WSN operations such as performing data aggregation among neighboring sensors. Note: Application layer doesn’t deal with WSN routing issues since they belong to Routing layer. It also doesn’t handle network congestion issues since they belong to Transport layer.

Although the above direct programming in application layer can perform many WSN data processing and other high-level wireless applications, it is still not convenient for a programmer to build those Application layer programs due to the following a few reasons:

1. Most WSN systems do not have convenient programming / compiling tools. For instance, NesC in TinyOS environment needs a long-time learning curve. A programmer needs to learn dozens of different entity interfaces. TinyOS installation is still a headache today.
More importantly, a programmer needs to get familiar with many WSN internal operation details in order to build an efficient, easy-to-use Application layer program. For instance, if a programmer wants to build sensor data query software (which is a basic WSN application layer function), she should know the routing layer details since it is the routing protocol that delivers query commands to each sensor. She should also understand the network topology since a data query command may need data from certain sensor areas.

Although a programmer can build Application layer programs to control a sensor’s behavior, many WSN operations are built on the collaboration of many nodes instead of just one node. For instance, to save wireless communication energy consumption, a data query command eventually collects data through data aggregation technology. That is, a program could reside in an individual node. But it needs to control many other nodes. Apparently, this is a challenging task for any programmer to achieve this.

Therefore, to unload the heavy responsibility of a WSN programmer, we need to add a new layer to traditional network stack. We call this new layer as Middleware layer. As shown in Figure 1 right part, the Middleware layer should be located between Application layer and Transport layer. Through the WSN middleware we could hide the complexity of low-level operations. A programmer can avoid the headache considerations of WSN dynamic network
topology and low-level embedded OS APIs (Application Program Interfaces).

A good WSN middleware provides a programmer some reusable code services which allow the programmer to access the functionality of network resources while minimizing the effort of dealing with code dissemination, data aggregation, and power management.

Although traditional middleware schemes (used in distributed computing systems) can also provide transparency abstractions by hiding the context information, they primarily aim to satisfy the interests of individual nodes. However, WSNs applications are data centric, and therefore middleware must be able to operate in all available nodes rather than for individual nodes. Moreover, WSNs middleware should support data aggregation in intermediate nodes along the forwarding path. But traditional distributed system middleware doesn’t need to support data aggregation since they use end-to-end paradigm. [Miaomiao08]

In middleware, data management is an important task. The middleware needs to provide appropriate abstractions of data structures and operations. Without such an abstraction, the application programmer must manage the heterogeneous data and low-level operations. [Miaomiao08]

When we design WSN middleware, we need to make sure that it is lightweight enough for implementation in sensor nodes with limited processing and energy resources.

Good Idea

How to minimize a programmer’s working load is the goal of many platforms. It is very time-consuming if a WSN programmer needs to understand all network operation details before she writes the Application layer program. Many WSN companies try to encapsulate complex sensor/network control into a set of APIs, which is part of middleware tasks. A programmer can then quickly come up with a useful application based on those friendly APIs.
7.2 A Reference Model of WSN Middleware [Miaomiao08]

7.2.1 Model Overview

As shown in Figure 7.2, WSN middleware includes 4 major components:

1. **Programming abstractions**: A middleware design should first define a set of friendly application program interfaces that hide all complex WSN operations.

2. **System services**: After defining program abstractions, a middleware should internally provide concrete implementations of these abstractions. Those implementations are called system services since they belong to part of system codes instead of user codes.

3. **Runtime support**: After we have the above system service codes, the sensor operating system (OS) should be able to run those codes in an optimized way. That is, we need to have runtime support.

4. **Quality-of-service (QoS)**: In the Application layer, people typically use QoS to define some visible application performance metrics such as data resolution, processing speed and network delay performance. The middleware should be able to adapt to different QoS requirements.

Place Figure 7.2 here.

**Figure 7.2** WSN Middleware Components [Miaomiao08]

The first 3 WSN middleware components have very close relationship among them. The purpose of defining programming abstractions is to hide complex WSN operations. From users’ viewpoint, they only require that the middleware provides a set of system services. From middleware designers’ viewpoint, they need to write codes to provide runtime support for those system services.
Figure 7.3 shows more details for each of the above components. Note that this is just a typical middleware reference model. It doesn’t mean that all WSN middleware implementation should include all of these components.

It is a misunderstanding that the middleware is only implemented in sensors. As a matter of fact, because a user can program the system in different places, the middleware can be located in a sensor node, a sink (i.e. a base station), a user terminal that communicates with a sink. The distributed middleware components in different places are able to communicate with each other to achieve common goals such as a data query execution. Figure 7.4 illustrates this point.

7.3 Middleware example: Agilla [CFok05] [Miaomiao08]

A type of middleware implementation is based on the concept of mobile agent, which is an execution thread that can migrate from a node to another. Such an agent encapsulates the
running codes, the system state and application data.

*Agilla* [CFok05] is an example of implementation on agent-based middleware. We can inject a new agent to the WSN in order to reprogram the network.

Figure 7.4 shows the *Agilla* system model. Note that each sensor node can support multiple agents. A node also maintains a *tuple space* and a *neighbor list*.

1. The *tuple space* can be shared by all agents residing on the same node. *Agilla* provides special instructions to remotely access the agents in another node’s tuple space.

2. The *neighbor list* contains the addresses of all directly adjacent nodes in the WSN. This is for the convenience of agent migration.

---

**Good Idea**

Mobile agent has attracted many researchers’ interests. Its basic feature is to allow a physical entity to transfer its unfinished task to another entity. Such a “chain” effect eventually achieves a systematic task. Please note that mobile agent concept is different from general multi-agent concept. The latter typically assumes that agents do not migrate / transfer between entities.

Place Figure 7.5 here.

**Figure 7.5** *Agilla* system model [CFok05]

As shown in Figure 7.6, a mobile agent in *Agilla* consists of a stack, a heap, and some registers. The heap is actually a memory space to store system variables. Similar to a common CPU architecture, a register consists of agent ID, program counter (PC), and the condition code. The agent ID is unique to each agent. The PC contains the address of the next instruction.
Place Figure 7.6 here.

**Figure 7.6** Agilla agent architecture [CFok05]

*Code migration* can be achieved by moving or cloning an agent from one node to another. A *tuple space* can package up all register variables during code migration. When an agent moves, it carries its state variable and runtime code. After an agent reaches a new node, it resumes the code executing. Multi-hop migration is handled by the middleware OS.

### 7.4 Middleware for Data Acquisition: Mires [ESouto04] [Miaomiao08]

*A typical task of WSN middleware is data management, which provides services to applications for data acquisition, data processing, and data storage.*

This section uses Mires [ESouto04] as an example of *data acquisition*, which includes a series of functions such as event definition, event registration/cancellation, event detection, and event delivery. Figure 7.7 shows Mires’ middleware architecture.

Place Figure 7.7 here.

**Figure 7.7** Mire’s architecture [ESouto04]

Mires uses publish/subscribe paradigm (see Figure 7.8) to implement event-based data
acquisition. Such a paradigm supports asynchronous communication and facilitates message exchange between the sensor nodes and the sink node. A publish/subscribe system has two basic components: the event subscriber (in the sink) and the event publisher (i.e. the event broker) (in the sensor nodes).

In Mires, the Application layer in the sink subscribes interested event data. Its subscribe messages are broadcasted down to the network nodes which publish their collected data to the network.

Place Figure 7.8 here.

**Figure 7.8** Mire’s Pub/Sub Component [ESouto04]

Middleware for query-based data model can use TinyDB [TinyDB]’s flooding approach to disseminate the queries throughout the network.

### 7.5 Data Storage: DSWare [SLi03] [Miaomiao08]

WSN middleware needs to support one of the most important tasks, i.e., data-centric storage. Data Service Middleware (DSWare) [SLi03] is such a middleware. As illustrated in Figure 7.9, DSWare implements a database-like abstraction composed of various data services.

1. The event detection component actually corresponds to the above-discussed data acquisition service.

2. The group management component can implement an important WSN feature, i.e., data aggregation.
(3) The *scheduling* component can schedule all middleware services based on two priorities: energy-efficiency or delay performance.

(4) The data storage component stores data according to the semantics associated with the data. It stores correlated data in geographically adjacent regions to achieve in-network processing.

(5) The caching component provides multiple copies of the data that requested most often. *DSWare* spreads the cached data over the network to achieve high availability and faster query execution.

Place Figure 7.9 here.

**Figure 7.9** DSWare components [SLi03]

### 7.6 WSN Runtime Support example: Mate [PLewis02] [Miaomiao08]

As we mentioned before, all defined middleware services should have some form of runtime support to ensure a well-defined execution environment.

The runtime support has the following basic functions: inter-process communication (IPC), memory control, and power management (in both voltage scaling and component deactivation). These functions are important since they can be used to implement higher-level middleware services such as multi-thread processing, smart task scheduling, memory access synchronization, and spread signal spectrum management.

Typically some kind of *virtual machine* is used to implement runtime support. Such a *virtual machine* is typically implemented as a *platform-specific* kernel on top of the embedded OS, but with platform-independent primitives for the generic WSN middleware services.
Mate [PLewis02] is such an example. It is built on top of TinyOS. Figure 7.10 illustrates Mate's architecture.

The core of the Mate architecture is a scheduler, which maintains a queue of run-able contexts and interleaves their execution. The Mate concurrency model is based on statically named resources such as shared variables, and operations must explicitly specify the shared resources they use.

Place Figure 7.10 here.

**Figure 7.10** The architecture of Mate [PLewis02] [Miaomiao08]

### 7.7 QoS Support example: MiLAN [WBHeinzelman04] [Miaomiao08]

QoS support is important for applications with the following requirements: fault-tolerance, reliability, security and real-time data processing. The following parameters can be used to express QoS in a WSN: packet delay, jitter and loss, throughput, and latency. However, we need more QoS metrics to make a quantitative performance measurement. For instance, we may define some new QoS parameters including data accuracy, aggregation delay, aggregation degree, coverage, and precision. A WSM middleware that provides QoS support can efficiently support data acquisition.

MiLAN [WBHeinzelman04] has defined a set of QoS support as shown in Fig.7.11. A WSN application program starts with the conveyance of a set of QoS parameters from the application layer to MiLAN (i.e., middleware layer). Such a QoS conveyance is achieved through a State-based Variable Requirements Graph and a Sensor QoS Graph.
(1) **State-based Variable Requirements Graph:** It specifies the application's minimum acceptable QoS for each performance parameter based on the current system running state.

(2) A **Sensor QoS Graph** is used to determine which sets of nodes a WSN may support from QoS requirements viewpoint.

Place Figure 7.11 here.

**Figure 7.11** QoS support in *MiLAN* [WBHeinzelman04]
Problems & Exercises

7.1 Explain the roles of WSN middleware in WSN system design. Point out its components and explain each of them briefly.

7.2 What challenges do we have when designing WSN middleware?

7.3 Why should we use programming abstractions?

7.4 How does Agilla implement code management?

7.5 Why does Mires use a publish/subscribe architecture?

7.6 How does DSWare handle data storage?

7.7 Explain MiLAN QoS support details.