PET ENGINEERING COLLEGE

An ISO 9001:2015 Certified Institution

Accredited by NAAC, Approved by AICTE, Recognized by Government of Tamil Nadu and Affiliated to Anna University

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT - V

SENSOR NETWORK PLATFORMS AND TOOLS

CLASS	**: S7 ECE**
SUBJECT CODE	**: EC8702**
SUBJECT NAME	**: ADHOC AND WIRELESS SENSOR NETWORKS**
REGULATION	**: 2017**

7 Sensor Network Platforms and Tools

In previous chapters, we discussed various aspects of sensor networks, including sensing and estimation, networking, infrastructure services, sensor tasking, and data storage and query. A real-world sensor network application most likely has to incorporate all these elements, subject to energy, bandwidth, computation, storage, and real-time constraints. This makes sensor network application development quite different from traditional distributed system development or database programming. With ad hoc deployment and frequently changing network topology, a sensor network application can hardly assume an always-on infrastructure that provides reliable services such as optimal routing, global directories, or service discovery.

There are two types of programming for sensor networks, those carried out by end users and those performed by application developers. An end user may view a sensor network as a pool of data and *interact* with the network via queries. Just as with query languages for database systems like SQL, a good sensor network programming language should be expressive enough to encode application logic at a high level of abstraction, and at the same time be structured enough to allow efficient execution on the distributed platform. Examples of sensor database query interfaces are described in Chapter 6. Ideally, the end users should be shielded away from details of how sensors are organized and how nodes communicate.

On the other hand, an application developer must provide end users of a sensor network with the capabilities of data acquisition, processing, and storage. Unlike general distributed or database systems, collaborative signal and information processing (CSIP) software comprises reactive, concurrent, distributed programs running on ad hoc, resource-constrained, unreliable computation and communication platforms. Developers at this level have to deal with all kinds of uncertainty in the real world. For example, signals are noisy, events can happen at the same time, communication and computation take time, communications may be unreliable, battery life is limited, and so on. Moreover, because of the amount of domain knowledge required, application developers are typically signal and information processing specialists, rather than operating systems and networking experts. How to provide appropriate programming abstractions to these application writers is a key challenge for sensor network software development. In this chapter, we focus on software design issues to support this type of programming.

To make our discussion of these software issues concrete, we first give an overview of a few representative sensor node hardware platforms (Section 7.1). In Section 7.2, we present the challenges of sensor network programming due to the massively concurrent interaction with the physical world. Section 7.3 describes TinyOS for Berkeley motes and two types of node-centric programming interfaces: an imperative language, nesC, and a dataflow-style language, TinyGALS. Node-centric designs are typically supported by node-level simulators such as ns-2 and TOSSIM, as described in Section 7.4. State-centric programming is a step toward programming beyond individual nodes. It gives programmers platform support for thinking in high-level abstractions, such as the state of the phenomena of interest over space and time. An example of state-centric platforms is given in Section 7.5.

7.1 Sensor Node Hardware

Sensor node hardware can be grouped into three categories, each of which entails a different set of trade-offs in the design choices.

- *Augmented general-purpose computers:* Examples include low-power PCs, embedded PCs (e.g., PC104), custom-designed PCs (e.g., Sensoria WINS NG nodes),[1] and various personal digital assistants (PDA). These nodes typically run off-the-shelf operating systems such as Win CE, Linux, or real-time operating systems and use standard wireless communication protocols such as Bluetooth or IEEE 802.11. Because of their relatively higher processing capability, they can accommodate a wide variety of sensors, ranging from simple microphones to more sophisticated video cameras.

Compared with dedicated sensor nodes, PC-like platforms are more power hungry. However, when power is not an issue, these platforms have the advantage that they can leverage the availability of fully supported networking protocols, popular programming languages, middleware, and other off-the-shelf software.

- *Dedicated embedded sensor nodes:* Examples include the Berkeley mote family [98], the UCLA Medusa family [202], Ember nodes,[2] and MIT μAMP [32]. These platforms typically use commercial off-the-shelf (COTS) chip sets with emphasis on small form factor, low power processing and communication, and simple sensor interfaces. Because of their COTS CPU, these platforms typically support at least one programming language, such as C. However, in order to keep the program footprint small to accommodate their small memory size, programmers of these platforms are given full access to hardware but barely any operating system support. A classical example is the TinyOS platform and its companion programming language, nesC. We will discuss these platforms in Sections 7.3.1 and 7.3.2.
- *System-on-chip (SoC) nodes:* Examples of SoC hardware include smart dust [109], the BWRC picoradio node [187], and the PASTA node.[3] Designers of these platforms try to push the hardware limits by fundamentally rethinking the hardware architecture trade-offs for a sensor node at the chip design level. The goal is to find new ways of integrating CMOS, MEMS, and RF technologies

1 See *http://www.sensoria.com/* and *http://www.janet.ucla.edu/WINS/*, and [158].

2 See *http://www.ember.com*.

3 See *http://pasta.east.isi.edu*.

to build extremely low power and small footprint sensor nodes that still provide certain sensing, computation, and communication capabilities. Since most of these platforms are currently in the research pipeline with no predefined instruction set, there is no software platform support available.

Among these hardware platforms, the Berkeley motes, due to their small form factor, open source software development, and commercial availability, have gained wide popularity in the sensor network research community. In the following section, we give an overview of the Berkeley MICA mote.

7.1.1 Berkeley Motes

The Berkeley motes are a family of embedded sensor nodes sharing roughly the same architecture. Figure 7.1 shows a comparison of a subset of mote types.

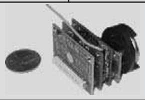
Mote type		WeC	Rene	Rene2	Mica	Mica2	Mica2Dot
Example picture							
MCU	Chip	AT90LS8535	ATmega163L		ATmega103L	ATmega128L	
	Type	4 MHz, 8 bit	4 MHz, 8 bit		4 MHz, 8 bit	8 MHz, 8 bit	
	Program memory (KB)	8	16		128	128	
	RAM (KB)	0.5	1		4	4	
External nonvolatile storage	Chip	24LC256			AT45DB014B		
	Connection type	I2C			SPI		
	Size (KB)	32			512		
Default power source	Type	Coin cell	2xAA				Coin cell
	Typical capacity (mAh)	575	2850				1000
RF	Chip	TR1000				CC1000	
	Radio frequency	868/916MHz				868/916MHz, 433, or 315 MHz	
	Raw speed (kbps)	10			40	38.4	
	Modulation type	On/Off key			Amplitude Shift key	Frequency Shift key	

Figure 7.1 A comparison of Berkeley motes.

Let us take the MICA mote as an example. The MICA motes have a two-CPU design, as shown in Figure 7.2. The main microcontroller (MCU), an Atmel ATmega103L, takes care of regular processing. A separate and much less capable coprocessor is only active when the MCU is being reprogrammed. The ATmega103L MCU has integrated 512 KB flash memory and 4 KB of data memory. Given these small memory sizes, writing software for motes is challenging. Ideally, programmers should be relieved from optimizing code at assembly level to keep code footprint small. However, high-level support and software services are not free. Being able to mix and match only necessary software components to support a particular application is essential to achieving a small footprint. A detailed discussion of the software architecture for motes is given in Section 7.3.1.

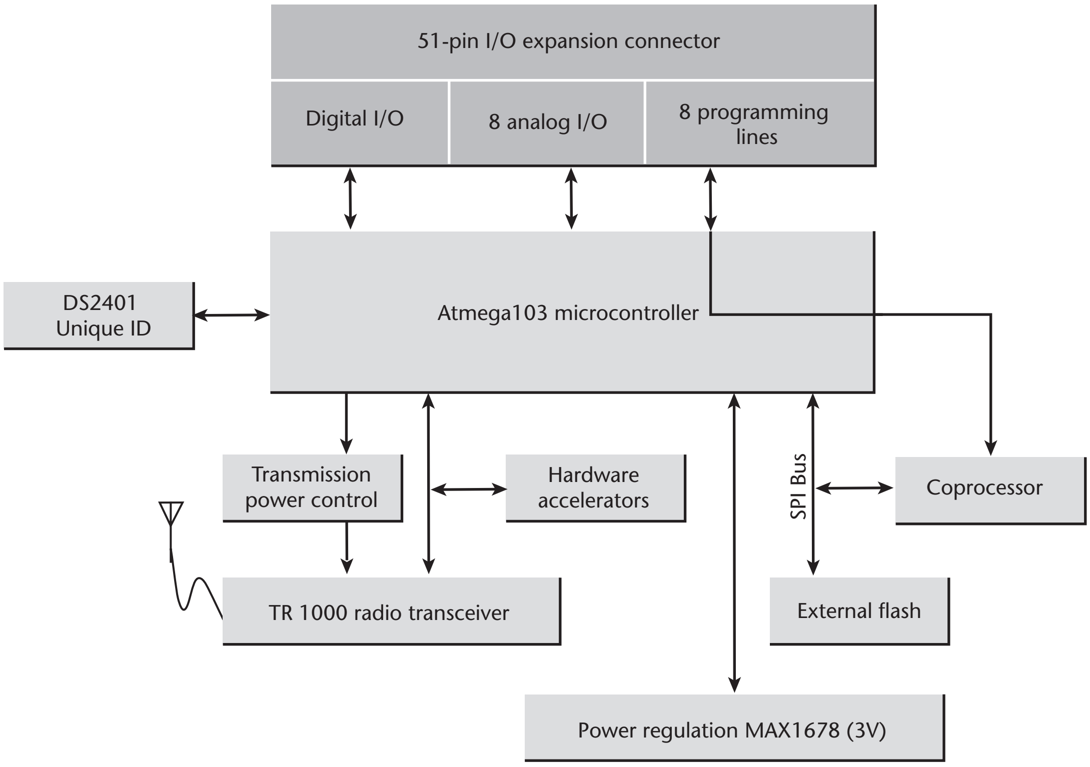

51-pin I/O expansion connector
Digital I/O
8 analog I/O
8 programming lines
DS2401 Unique ID
Atmega103 microcontroller
Transmission power control
Hardware accelerators
SPI Bus
Coprocessor
TR 1000 radio transceiver
External flash
Power regulation MAX1678 (3V)

Figure 7.2 MICA mote architecture.

In addition to the memory inside the MCU, a MICA mote also has a separate 512 KB flash memory unit that can hold data. Since the connection between the MCU and this external memory is via a low-speed serial peripheral interface (SPI) protocol, the external memory is more suited for storing data for later batch processing than for storing programs. The RF communication on MICA motes uses the TR1000 chip set (from RF Monolithics, Inc.) operating at 916 MHz band. With hardware accelerators, it can achieve a maximum of 50 kbps raw data rate. MICA motes implement a 40 kbps transmission rate. The transmission power can be digitally adjusted by software though a potentiometer (Maxim DS1804). The maximum transmission range is about 300 feet in open space.

Like other types of motes in the family, MICA motes support a 51 pin I/O extension connector. Sensors, actuators, serial I/O boards, or parallel I/O boards can be connected via the connector. A sensor/actuator board can host a temperature sensor, a light sensor, an accelerometer, a magnetometer, a microphone, and a beeper. The serial I/O (UART) connection allows the mote to communicate with a PC in real time. The parallel connection is primarily for downloading programs to the mote.

It is interesting to look at the energy consumption of various components on a MICA mote. As shown in Figure 7.3, a radio

Component	*Rate*	*Startup time*	*Current consumption*
MCU active	4 MHz	N/A	5.5 mA
MCU idle	4 MHz	1 μs	1.6 mA
MCU suspend	32 kHz	4 ms	<20 μA
Radio transmit	40 kHz	30 ms	12 mA
Radio receive	40 kHz	30 ms	1.8 mA
Photoresister	2000 Hz	10 ms	1.235 mA
Accelerometer	100 Hz	10 ms	5 mA/axis
Temperature	2 Hz	500 ms	0.150 mA

Figure 7.3 Power consumption of MICA motes.

transmission bears the maximum power consumption. However, each radio packet (e.g., 30 bytes) only takes 4 ms to send, while listening to incoming packets turns the radio receiver on all the time. The energy that can send one packet only supports the radio receiver for about 27 ms. Another observation is that there are huge differences among the power consumption levels in the active mode, the idle mode, and the suspend mode of the MCU. It is thus worthwhile from an energy-saving point of view to suspend the MCU and the RF receiver as long as possible.

7.2 Sensor Network Programming Challenges

Traditional programming technologies rely on operating systems to provide abstraction for processing, I/O, networking, and user interaction hardware, as illustrated in Figure 7.4. When applying such a

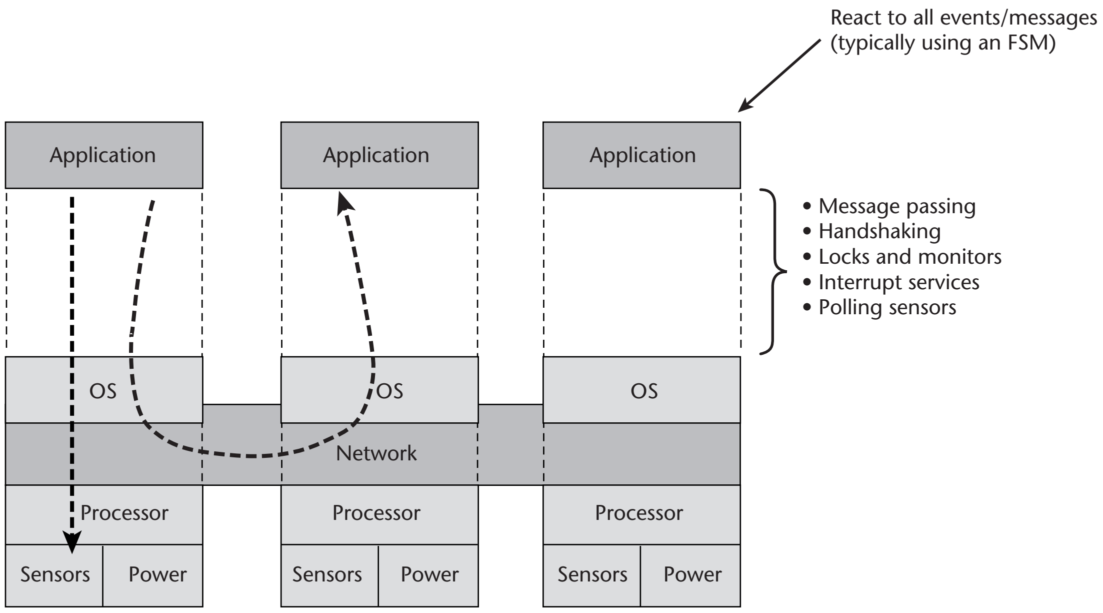


Figure 7.4 Traditional embedded system programming interface.

model to programming networked embedded systems, such as sensor networks, the application programmers need to explicitly deal with message passing, event synchronization, interrupt handing, and sensor reading. As a result, an application is typically implemented as a finite state machine (FSM) that covers all extreme cases: unreliable communication channels, long delays, irregular arrival of messages, simultaneous events, and so on. In a target tracking application implemented on a Linux operating system and with directed diffusion routing, roughly 40 percent of the code implements the FSM and the glue logic of interfacing computation and communication [142].

For resource-constrained embedded systems with real-time requirements, several mechanisms are used in embedded operating systems to reduce code size, improve response time, and reduce energy consumption. Microkernel technologies [211] modularize the operating system so that only the necessary parts are deployed with the application. Real-time scheduling [27] allocates resources to more urgent tasks so that they can be finished early. Event-driven execution allows the system to fall into low-power sleep mode when no interesting events need to be processed. At the extreme, embedded operating systems tend to expose more hardware controls to the programmers, who now have to directly face device drivers and scheduling algorithms, and optimize code at the assembly level. Although these techniques may work well for small, stand-alone embedded systems, they do not scale up for the programming of sensor networks for two reasons.

- Sensor networks are large-scale distributed systems, where global properties are derivable from program execution in a massive number of distributed nodes. Distributed algorithms themselves are hard to implement, especially when infrastructure support is limited due to the ad hoc formation of the system and constrained power, memory, and bandwidth resources.
- As sensor nodes deeply embed into the physical world, a sensor network should be able to respond to multiple concurrent stimuli at the speed of changes of the physical phenomena of interest.

In the rest of the chapter, we give several examples of sensor network software design platforms. We discuss them in terms of both *design methodologies* and *design platforms*. A design methodology implies a conceptual model for programmers, with associated techniques for problem decomposition for the software designers. For example, does the programmer think in terms of events, message passing, and synchronization, or does he/she focus more on information architecture and data semantics? A design platform supports a design methodology by providing design-time (precompile time) language constructs and restrictions, and run-time (postcompile time) execution services.

There is no single universal design methodology for all applications. Depending on the specific tasks of a sensor network and the way the sensor nodes are organized, certain methodologies and platforms may be better choices than others. For example, if the network is used for monitoring a small set of phenomena and the sensor nodes are organized in a simple star topology, then a client-server software model would be sufficient. If the network is used for monitoring a large area from a single access point (i.e., the base station), and if user queries can be decoupled into aggregations of sensor readings from a subset of sensor nodes, then a tree structure that is rooted at the base station is a better choice. However, if the phenomena to be monitored are moving targets, as in the target tracking examples discussed in Chapter 2, then neither the simple client-server model nor the tree organization is optimal. More sophisticated design methodologies and platforms are required.

7.3 Node-Level Software Platforms

Most design methodologies for sensor network software are node-centric, where programmers think in terms of how a node should behave in the environment. A node-level platform can be a node-centric operating system, which provides hardware and networking abstractions of a sensor node to programmers, or it can be a language platform, which provides a library of components to programmers.

A typical operating system abstracts the hardware platform by providing a set of services for applications, including file management, memory allocation, task scheduling, peripheral device drivers, and networking. For embedded systems, due to their highly specialized applications and limited resources, their operating systems make different trade-offs when providing these services. For example, if there is no file management requirement, then a file system is obviously not needed. If there is no dynamic memory allocation, then memory management can be simplified. If prioritization among tasks is critical, then a more elaborate priority scheduling mechanism may be added.

TinyOS [98] and TinyGALS [38] are two representative examples of node-level programming tools that we will cover in detail in this section. Other related software platforms include Maté [130], a virtual machine for the Berkeley motes. Observing that operations such as polling sensors and accessing internal states are common to all sensor network application, Maté defines virtual machine instructions to abstract those operations. When a new hardware platform is introduced with support for the virtual machine, software written in the Maté instruction set does not have to be rewritten.

7.3.1 Operating System: TinyOS

TinyOS aims at supporting sensor network applications on resource-constrained hardware platforms, such as the Berkeley motes.

To ensure that an application code has an extremely small footprint, TinyOS chooses to have no file system, supports only static memory allocation, implements a simple task model, and provides minimal device and networking abstractions. Furthermore, it takes a language-based application development approach, to be discussed later, so that only the necessary parts of the operating system are compiled with the application. To a certain extent, each TinyOS application is built into the operating system.

Like many operating systems, TinyOS organizes components into layers. Intuitively, the lower a layer is, the "closer" it is to the hardware; the higher a layer is, the "closer" it is to the application.

In addition to the layers, TinyOS has a unique component architecture and provides as a library a set of system software components. A component specification is independent of the component implementation. Although most components encapsulate software functionalities, some are just thin wrappers around hardware. An application, typically developed in the nesC language covered in the next section, *wires* these components together with other application-specific components.

Let us consider a TinyOS application example—`FieldMonitor`, where all nodes in a sensor field periodically send their temperature and photo sensor readings to a base station via an ad hoc routing mechanism. A diagram of the `FieldMonitor` application is shown in Figure 7.5, where blocks represent TinyOS components and arrows represent function calls among them. The directions of the arrows are from callers to callees.

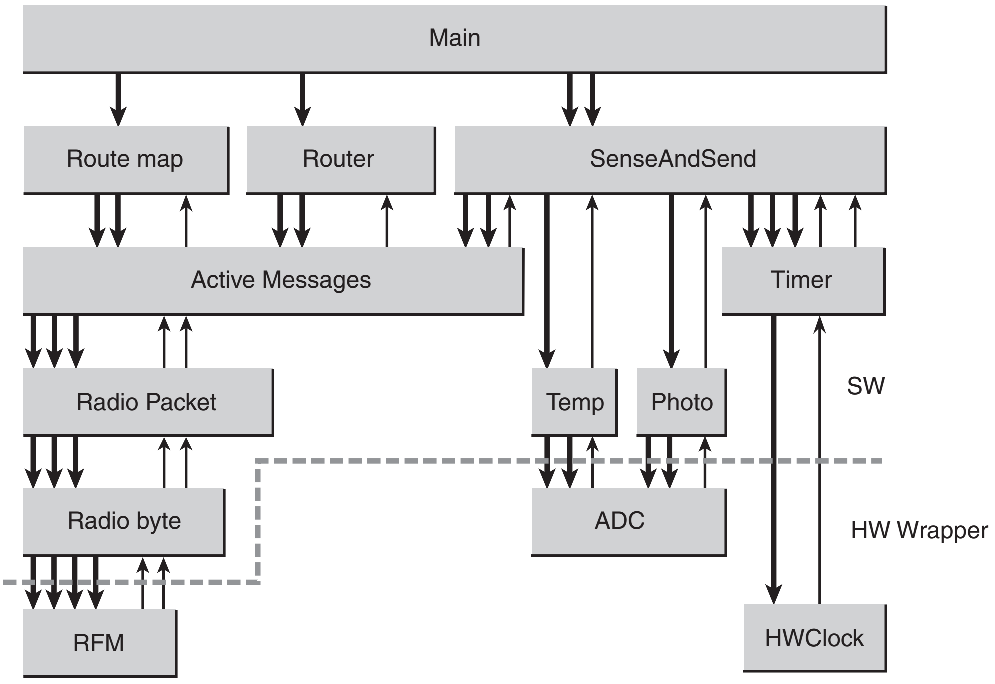


Figure 7.5 The `FieldMonitor` application for sensing and sending measurements.

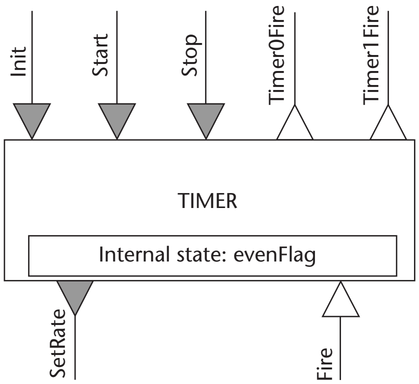


Figure 7.6 The Timer component and its interfaces.

To explain in detail the semantics of TinyOS components, let us first look at the Timer component of the FieldMonitor application, as shown in Figure 7.6. This component is designed to work with a clock, which is a software wrapper around a hardware clock that generates periodic interrupts. The method calls of the Timer component are shown in the figure as the arrowheads. An arrowhead pointing into the component is a method of the component that other components can call. An arrowhead pointing outward is a method that this component requires another layer component to provide. The absolute directions of the arrows, up or down, illustrate this component's relationship with other layers. For example, the Timer depends on a lower layer HWClock component. The Timer can set the rate of the clock, and in response to each clock interrupt it toggles an internal Boolean flag, evenFlag, between true (or 1) and false (or 0). If the flag is 0, the Timer produces a timer0Fire event to trigger other components; otherwise, it produces a timer1Fire event. The Timer has an init() method that initializes its internal flag, and it can be enabled and disabled via the start and stop calls.

A program executed in TinyOS has two contexts, *tasks* and *events*, which provide two sources of concurrency. Tasks are created (also called *posted*) by components to a task scheduler. The default implementation of the TinyOS scheduler maintains a task queue and invokes tasks according to the order in which they were posted. Thus tasks are deferred computation mechanisms. Tasks always run to completion without preempting or being preempted by other tasks. Thus tasks are nonpreemptive. The scheduler invokes a new task from the task queue only when the current task has completed. When no tasks are available in the task queue, the scheduler puts the CPU into the sleep mode to save energy.

The ultimate sources of triggered execution are events from hardware: clock, digital inputs, or other kinds of interrupts. The execution of an interrupt handler is called an *event context*. The processing of events also runs to completion, but it preempts tasks and can be preempted by other events. Because there is no preemption mechanism among tasks and because events always preempt tasks, programmers are required to chop their code, especially the code in the event contexts, into small execution pieces, so that it will not block other tasks for too long.

Another trade-off between nonpreemptive task execution and program reactiveness is the design of split-phase operations in TinyOS. Similar to the notion of asynchronous method calls in distributed computing, a split-phase operation separates the initiation of a method call from the return of the call. A call to a split-phase operation returns immediately, without actually performing the body of the operation. The true execution of the operation is scheduled later; when the execution of the body finishes, the operation notifies the original caller through a separate method call. An example of a split-phase operation is the packet send method in the `Active Messages` (AM) component, used in Figure 7.5. Sending a packet is a long operation, involving converting the packets to bytes, then to bits, and ultimately driving the RF circuits to send the bits one by one. Without a split-phase execution, sending a packet will block the entire system from reacting to new events for a significant period of time. In the TinyOS implementation, the `send()` command in the AM component returns immediately. However, it is the caller's responsibility to remember that the packet has not yet been sent. When the packet is indeed sent, the AM component will notify its caller by a `sendDone()` method call. Only at this time is the AM component ready to accept another packet.

In TinyOS, resource contention is typically handled through explicit rejection of concurrent requests. All split-phase operations return Boolean values indicating whether a request to perform the operation is accepted. In the above example, a call of `send()`, when the AM component is still sending the first packet, will result in an error signaled by the AM component. To avoid such an error, the caller of the AM component typically implements a *pending* lock, to remember not to request further sendings until the `sendDone()` method is called. To avoid loss of packets, a queue should be incorporated by the caller if necessary.

In summary, many design decisions in TinyOS are made to ensure that it is extremely lightweight. Using a component architecture that contains all variables inside the components and disallowing dynamic memory allocation reduces the memory management overhead and makes the data memory usage statically analyzable. The simple concurrency model allows high concurrency with low thread maintenance overhead. As a consequence, the entire `FieldMonitor` system shown in Figure 7.5 takes only 3 KB of space for code and 226 bytes for data. However, the advantage of being lightweight is not without cost. Many hardware idiosyncrasies and complexities of concurrency management are left for the application programmers to handle. Several tools have been developed to give programmers language-level support for improving programming productivity and code robustness. We introduce in the next two sections two special-purpose languages for programming sensor network nodes. Although both languages are designed on top of TinyOS, the principles they represent may apply to other platforms.

7.3.2 Imperative Language: nesC

nesC [79] is an extension of C to support and reflect the design of TinyOS v1.0 and above. It provides a set of language constructs and restrictions to implement TinyOS components and applications.

Component Interface

A component in nesC has an interface specification and an implementation. To reflect the layered structure of TinyOS, interfaces of a nesC component are classified as *provides* or *uses* interfaces. A provides interface is a set of method calls exposed to the upper layers, while a uses interface is a set of method calls hiding the lower layer components. Methods in the interfaces can be grouped and named. For example, the interface specification of the `Timer` component in Figure 7.6 is listed in Figure 7.7. The interface, again, independent of the implementation, is called `TimerModule`.

Although they have the same method call semantics, nesC distinguishes the *directions* of the interface calls between layers as *event* calls

```
module TimerModule {
  provides {
    interface StdControl;
    interface Timer01;
  }
  uses interface Clock as Clk;
}

interface StdControl {
  command result_t init();
}

interface Timer01 {
  command result_t start(char type, uint32_t interval;
  command result_t stop();
  event result_t timer0Fire();
  event result_t timer1Fire();
}

interface Clock {
  command result_t setRate(char interval, char scale);
  event result_t fire();
}
```

Figure 7.7 The interface definition of the `Timer` component in nesC.

and *command* calls. An event call is a method call from a lower layer component to a higher layer component, while a command is the opposite. Note that one needs to know both the type of the interface (provides or uses) and the direction of the method call (event or command) to know exactly whether an interface method is implemented by the component or is required by the component.

The separation of interface type definitions from how they are used in the components promotes the reusability of standard interfaces. A component can provide and use the same interface type, so that it can act as a filter interposed between a client and a service. A component may even use or provide the same interface multiple times. In these cases, the component must give each interface instance a separate name using the `as` notation, as shown in the `Clock` interface in Figure 7.7.

Component Implementation

There are two types of components in nesC, depending on how they are implemented: *modules* and *configurations*. Modules are implemented by application code (written in a C-like syntax). Configurations are implemented by connecting interfaces of existing components.

The implementation part of a module is written in C-like code. A command or an event `bar` in an interface `foo` is referred as `foo.bar`. A keyword `call` indicates the invocation of a command. A keyword `signal` indicates the triggering by an event. For example, Figure 7.8 shows part of the implementation of the Timer component, whose interface is defined in Figure 7.7. In a sense, this implementation is very much like an object in object-oriented programming without any constructors.

Configuration is another kind of implementation of components, obtained by connecting existing components. Suppose we want to connect the `Timer` component and a hardware clock wrapper, called `HWClock`, to provide a timer service, called `TimerC`. Figure 7.9 shows a conceptual diagram of how the components are connected, and Figure 7.10 shows the corresponding nesC code.

```
module Timer {
  provides {
    interface StdControl;
    interface Timer01;
  }
  uses interface Clock as Clk;
}
implementation {
  bool evenFlag;

  command result_t StdControl.init() {
    evenFlag = 0;
    return call Clk.setRate(128, 4); //4 ticks per second
  }

  event result_t Clk.fire() {
    evenFlag = !evenFlag;
    if (evenFlag) {
      signal Timer01.timer0Fire();
    } else {
      signal Timer01.timer1Fire();
    }
    return SUCCESS;
  }
  ...
}
```

Figure 7.8 The implementation definition of the `Timer` component in nesC.

First of all, notice that the keyword `configuration` in the specification indicates that this component is not implemented directly as a module. In the implementation section of the configuration, the code first includes the two components, and then specifies that the interface `StdControl` of the `TimerC` component is the `StdControl` interface of the `TimerModule`; similarly for the `Timer01` interface. The connection between the `Clock` interfaces is specified using the `->` operator. Essentially, this interface is hidden from upper layers.

nesC also supports the creation of several instances of a component by declaring *abstract components* with optional parameters. Abstract components are created at compile time in configurations.

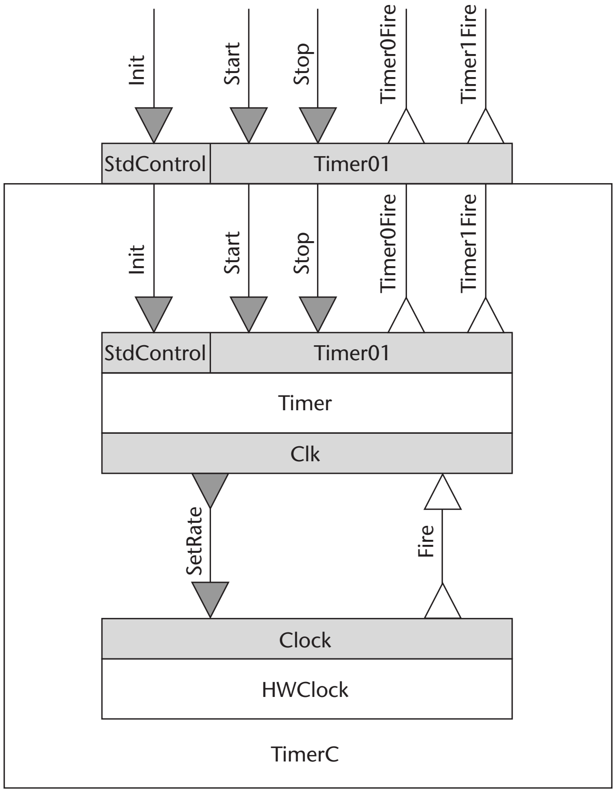


Figure 7.9 The `TimerC` configuration implemented by connecting `Timer` with `HWClock`.

Recall that TinyOS does not support dynamic memory allocation, so all components are statically constructed at compile time.

A complete application is always a configuration rather than a module. An application must contain the `Main` module, which links the code to the scheduler at run time. The `Main` has a single `StdControl` interface, which is the ultimate source of initialization of all components.

Concurrency and Atomicity

The language nesC directly reflects the TinyOS execution model through the notion of command and event contexts. Figure 7.11

```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer01;
  }
}
implementation {
  components TimerModule, Clock;

  StdControl = TimerModule.StdControl;
  Timer = TimerModule.Timer;

  TimerModule.Clk -> HWClock.Clock;
}
```

Figure 7.10 The implementation definition of the `TimerC` configuration in nesC.

shows a section of the component `SenseAndSend` to illustrate some language features to support concurrency in nesC and the effort to reduce race conditions. The `SenseAndSend` component is intended to be built on top of the `Timer` component (described in the previous section), an ADC component, which can provide sensor readings, and a communication component, which can send (or, more precisely, broadcast) a packet. When responding to a `timer0Fire` event, the `SenseAndSend` component invokes the ADC to poll a sensor reading. Since polling a sensor reading can take a long time, a split-phase operation is implemented for getting sensor readings. The call to `ADC.getData()` returns immediately, and the completion of the operation is signaled by an `ADC.dataReady()` event. A busy flag is used to explicitly reject new requests while the ADC is fulfilling an existing request. The `ADC.getData()` method sets the flag to true, while the `ADC.dataReady()` method sets it back to false. Sending the sensor reading to the next-hop neighbor via wireless communication is also a long operation. To make sure that it does not block the processing of the `ADC.dataReady()` event, a separate task is posted to the scheduler. A task is a method defined using the `task` keyword. In order

```
module SenseAndSend{
  provides interface StdControl;
    uses interface ADC;
    uses interface Timer:
    uses interface Send;
}

implementation {
  bool busy;
  norace uint16_t sensorReading;

  command result_t StdControl.init() {
    busy = FALSE;
  }

  event result_t Timer.timer0Fire() {
    bool localBusy;
    atomic {
      localBusy = busy;
      busy = TRUE;
    }
    if (!localBusy} {
      call ADC.getData(); //start getting sensor reading
      return SUCESS;
    } else {
      return FAILED;
    }
  }

  task void sendData() { // send sensorReading
    adcPacket.data = sensorReading;
    call Send.send(&adcPacket, sizeof adcPacket.data};
    return SUCESS;
  }

  event result_t ADC.dataReady(uinit16_t data) {
    sensorReading = data;
    post sendData();
    atomic {
      busy = FALSE;
    }
    return SUCCESS;
  }
  ...
}
```

Figure 7.11 A section of the implementation of `SenseAndSend`, illustrating the handling of concurrency in nesC.

to simplify the data structures inside the scheduler, a task cannot have arguments. Thus the sensor reading to be sent is put into a `sensorReading` variable.

There is one source of race condition in the `SenseAndSend`, which is the updating of the `busy` flag. To prevent some state from being updated by both scheduled tasks and event-triggered interrupt handlers, nesC provides language facilities to limit the race conditions among these operations.

In nesC, code can be classified into two types:

- *Asynchronous code (AC):* Code that is reachable from at least one interrupt handler.
- *Synchronous code (SC):* Code that is only reachable from tasks.

Because the execution of TinyOS tasks are nonpreemptive and interrupt handlers preempts tasks, SC is always atomic with respect to other SCs. However, any update to shared state from AC, or from SC that is also updated from AC, is a potential race condition. To reinstate atomicity of updating shared state, nesC provides a keyword `atomic` to indicate that the execution of a block of statements should not be preempted. This construction can be efficiently implemented by turning off hardware interrupts. To prevent blocking the interrupts for too long and affecting the responsiveness of the node, nesC does not allow method calls in atomic blocks. In fact, nesC has a compiler rule to enforce the accessing of shared variables to maintain the race-free condition. If a variable x is accessed by AC, then any access of x outside of an atomic statement is a compile-time error. This rule may be too rigid in reality. When a programmer knows for sure that a data race is not going to occur, or does not care if it occurs, then a `norace` declaration of the variable can prevent the compiler from checking the race condition on that variable.

Thus, to correctly handle concurrency, nesC programmers need to have a clear idea of what is synchronous code and what is asynchronous code. However, since the semantics is hidden away in the layered structure of TinyOS, it is sometimes not obvious to the programmers where to add atomic blocks.

along with a buffer for the storage location. The pair of access functions consists of a `PARAM_GET()` function that returns the value of the global variable, and a `PARAM_PUT()` function that stores a new value for the variable in the variable's buffer. A generated flag indicates whether the scheduler needs to update the variables by copying data from the buffer.

Since most of the data structures in the TinyGALS run-time scheduler are generated, the scheduler does not need to worry about handling different data types and the conversion among them. What is left in the run-time scheduler is merely event-queuing and function-triggering mechanisms. As a result, the TinyGALS run-time scheduler is very lightweight. The scheduler itself takes 112 bytes of memory, comparable with the original 86-byte TinyOS v0.6.1 scheduler.

7.4 Node-Level Simulators

Node-level design methodologies are usually associated with simulators that simulate the behavior of a sensor network on a per-node basis. Using simulation, designers can quickly study the performance (in terms of timing, power, bandwidth, and scalability) of potential algorithms without implementing them on actual hardware and dealing with the vagaries of actual physical phenomena.

A node-level simulator typically has the following components:

- *Sensor node model:* A node in a simulator acts as a software execution platform, a sensor host, as well as a communication terminal. In order for designers to focus on the application-level code, a node model typically provides or simulates a communication protocol stack, sensor behaviors (e.g., sensing noise), and operating system services. If the nodes are mobile, then the positions and motion properties of the nodes need to be modeled. If energy characteristics are part of the design considerations, then the power consumption of the nodes needs to be modeled.

- *Communication model:* Depending on the details of modeling, communication may be captured at different layers. The most elaborate simulators model the communication media at the physical layer, simulating the RF propagation delay and collision of simultaneous transmissions. Alternately, the communication may be simulated at the MAC layer or network layer, using, for example, stochastic processes to represent low-level behaviors.

- *Physical environment model:* A key element of the environment within which a sensor network operates is the physical phenomenon of interest. The environment can also be simulated at various levels of detail. For example, a moving object in the physical world may be abstracted into a point signal source. The motion of the point signal source may be modeled by differential equations or interpolated from a trajectory profile. If the sensor network is passive—that is, it does not impact the behavior of the environment—then the environment can be simulated separately or can even be stored in data files for sensor nodes to read in. If, in addition to sensing, the network also performs actions that influence the behavior of the environment, then a more tightly integrated simulation mechanism is required.
- *Statistics and visualization:* The simulation results need to be collected for analysis. Since the goal of a simulation is typically to derive global properties from the execution of individual nodes, visualizing global behaviors is extremely important. An ideal visualization tool should allow users to easily observe on demand the spatial distribution and mobility of the nodes, the connectivity among nodes, link qualities, end-to-end communication routes and delays, phenomena and their spatio-temporal dynamics, sensor readings on each node, sensor node states, and node lifetime parameters (e.g., battery power).

A sensor network simulator simulates the behavior of a subset of the sensor nodes with respect to time. Depending on how the time is advanced in the simulation, there are two types of execution models: *cycle-driven simulation* and *discrete-event simulation*. A cycle-driven (CD) simulation discretizes the continuous notion of real time into (typically regularly spaced) ticks and simulates the system behavior at these ticks. At each tick, the physical phenomena are first simulated, and then all nodes are checked to see if they have anything to sense, process, or communicate. Sensing and computation are assumed to be finished before the next tick. Sending a packet is also assumed to be completed by then. However, the packet will not be available for the destination node until the next tick. This split-phase communication is a key mechanism to reduce cyclic dependencies that may occur in cycle-driven simulations. That is, there should be no two components, such that one of them computes $y_k = f(x_k)$ and the other computes $x_k = g(y_k)$, for the same tick index k. In fact, one of the most subtle issues in designing a CD simulator is how to detect and deal with cyclic dependencies among nodes or algorithm components. Most CD simulators do not allow interdependencies within a single tick. Synchronous languages [91], which are typically used in control system designs rather than sensor network designs, do allow cyclic dependencies. They use a fixed-point semantics to define the behavior of a system at each tick.

Unlike cycle-driven simulators, a discrete-event (DE) simulator assumes that the time is continuous and an event may occur at any time. An event is a 2-tuple with a value and a time stamp indicating when the event is supposed to be handled. Components in a DE simulation react to input events and produce output events. In node-level simulators, a component can be a sensor node and the events can be communication packets; or a component can be a software module within a node and the events can be message passings among these modules. Typically, components are *causal*, in the sense that if an output event is computed from an input event, then the time stamp of the output event should not be earlier than that of the input event. Noncausal components require the simulators to be able to roll back in time, and, worse, they may not define a deterministic behavior of a system [129]. A DE simulator typically requires a global event queue. All events passing between nodes or modules are put in the event queue and sorted according to their chronological order. At each iteration of the simulation, the simulator removes the first event (the one with the earliest time stamp) from the queue and triggers the component that reacts to that event.

In terms of timing behavior, a DE simulator is more accurate than a CD simulator, and, as a consequence, DE simulators run slower. The overhead of ordering all events and computation, in addition to the values and time stamps of events, usually dominates the computation time. At an early stage of a design when only the asymptotic behaviors rather than timing properties are of concern, CD simulations usually require less complex components and give faster simulations. Partly because of the approximate timing behaviors, which make simulation results less comparable from application to application, there is no general CD simulator that fits all sensor network simulation tasks. We have come across a number of homegrown simulators written in Matlab, Java, and C++. Many of them are developed for particular applications and exploit application-specific assumptions to gain efficiency.

DE simulations are sometimes considered as good as actual implementations, because of their continuous notion of time and discrete notion of events. There are several open-source or commercial simulators available. One class of these simulators comprises extensions of classical network simulators, such as ns-2,[6] J-Sim (previously known as JavaSim),[7] and GloMoSim/QualNet.[8] The focus of these simulators is on network modeling, protocols stacks, and simulation performance. Another class of simulators, sometimes called *software-in-the-loop simulators*, incorporate the actual node software into the simulation. For this reason, they are typically attached to particular hardware platforms and are less portable. Examples include TOSSIM [131] for Berkeley motes and Em* (pronounced *em star*) [62] for Linux-based nodes such as Sensoria WINS NG platforms.

7.4.1 The ns-2 Simulator and its Sensor Network Extensions

The simulator ns-2 is an open-source network simulator that was originally designed for wired, IP networks. Extensions have been made

6 Available at *http://www.isi.edu/nsnam/ns*.
7 Available at *http://www.j-sim.org*.
8 Available at *http://pcl.cs.ucla.edu/projects/glomosim*.

to simulate wireless/mobile networks (e.g., 802.11 MAC and TDMA MAC) and more recently sensor networks. While the original ns-2 only supports logical addresses for each node, the wireless/mobile extension of it (e.g., [25]) introduces the notion of node locations and a simple wireless channel model. This is not a trivial extension, since once the nodes move, the simulator needs to check for each physical layer event whether the destination node is within the communication range. For a large network, this significantly slows down the simulation speed.

There are at least two efforts to extend ns-2 to simulate sensor networks: SensorSim from UCLA[9] and the NRL sensor network extension from the Navy Research Laboratory.[10] SensorSim aims at providing an energy model for sensor nodes and communication, so that power properties can be simulated [175]. SensorSim also supports hybrid simulation, where some real sensor nodes, running real applications, can be executed together with a simulation. The NRL sensor network extension provides a flexible way of modeling physical phenomena in a discrete event simulator. Physical phenomena are modeled as network nodes which communicate with real nodes through physical layers. Any interesting events are sent to the nodes that can sense them as a form of communication. The receiving nodes simply have a sensor stack parallel to the network stack that processes these events.

The main functionality of ns-2 is implemented in C++, while the dynamics of the simulation (e.g., time-dependent application characteristics) is controlled by Tcl scripts. Basic components in ns-2 are the layers in the protocol stack. They implement the *handlers* interface, indicating that they handle events. Events are communication packets that are passed between consecutive layers within one node, or between the same layers across nodes.

The key advantage of ns-2 is its rich libraries of protocols for nearly all network layers and for many routing mechanisms. These protocols

9 Available at *http://nesl.ee.ucla.edu/projects/sensorsim/*.

10 Available at *http://pf.itd.nrl.navy.mil/projects/nrlsensorsim/*.

are modeled in fair detail, so that they closely resemble the actual protocol implementations. Examples include the following:

- TCP: reno, tahoe, vegas, and SACK implementations
- MAC: 802.3, 802.11, and TDMA
- Ad hoc routing: Destination sequenced distance vector (DSDV) routing, dynamic source routing (DSR), ad hoc on-demand distance vector (AODV) routing, and temporally ordered routing algorithm (TORA)
- Sensor network routing: Directed diffusion, geographical routing (GEAR) and geographical adaptive fidelity (GAF) routing.

7.4.2 The Simulator TOSSIM

TOSSIM is a dedicated simulator for TinyOS applications running on one or more Berkeley motes. The key design decisions on building TOSSIM were to make it scalable to a network of potentially thousands of nodes, and to be able to use the actual software code in the simulation. To achieve these goals, TOSSIM takes a cross-compilation approach that compiles the nesC source code into components in the simulation. The event-driven execution model of TinyOS greatly simplifies the design of TOSSIM. By replacing a few low-level components, such as the A/D conversion (ADC), the system clock, and the radio front end, TOSSIM translates hardware interrupts into discrete-event simulator events. The simulator event queue delivers the interrupts that drive the execution of a node. The upper-layer TinyOS code runs unchanged.

TOSSIM uses a simple but powerful abstraction to model a wireless network. A network is a *directed* graph, where each vertex is a sensor node and each directed edge has a bit-error rate. Each node has a private piece of state representing what it hears on the radio channel. By setting connections among the vertices in the graph and a bit-error rate on each connection, wireless channel characteristics, such as imperfect channels, hidden terminal problems, and asymmetric links, can be easily modeled. Wireless transmissions are simulated at the bit level. If a bit error occurs, the simulator flips the bit.

TOSSIM has a visualization package called TinyViz, which is a Java application that can connect to TOSSIM simulations. TinyViz also provides mechanisms to control a running simulation by, for example, modifying ADC readings, changing channel properties, and injecting packets. TinyViz is designed as a communication service that interacts with the TOSSIM event queue. The exact visual interface takes the form of plug-ins that can interpret TOSSIM events. Beside the default visual interfaces, users can add application-specific ones easily.

7.5 Programming Beyond Individual Nodes: State-Centric Programming

Many sensor network applications, such as target tracking, are not simply generic distributed programs over an ad hoc network of energy-constrained nodes. Deeply rooted in these applications is the notion of states of physical phenomena and models of their evolution over space and time. Some of these states may be represented on a small number of nodes and evolve over time, as in the target tracking problem in Chapter 2, while others may be represented over a large and spatially distributed number of nodes, as in tracking a temperature contour.

A distinctive property of physical states, such as location, shape, and motion of objects, is their continuity in space and time. Their sensing and control is typically done through sequential state updates. System theories, the basis for most signal and information processing algorithms, provide abstractions for state update, such as:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, u_k) \tag{7.1}$$

$$\mathbf{y}_k = g(\mathbf{x}_k, u_k) \tag{7.2}$$

where $\mathbf{x}$ is the state of a system, u are the inputs, $\mathbf{y}$ are the outputs, k is an integer update index over space and/or time, f is the state update function, and g is the output or observation function. This formulation is broad enough to capture a wide variety of algorithms in sensor fusion, signal processing, and control (e.g., Kalman filtering, Bayesian estimation, system identification, feedback control laws, and finite-state automata).

However, in distributed real-time embedded systems such as sensor networks, the formulation is not so clean as represented in those equations. The relationships among subsystems can be highly complex and dynamic over space and time. The following concerns, not explicitly raised (7.1) and (7.2), must be properly addressed during the design to ensure the correctness and efficiency of the resulting systems.

- Where are the state variables stored?
- Where do the inputs come from?
- Where do the outputs go?
- Where are the functions f and g evaluated?
- How long does the acquisition of inputs take?
- Are the inputs in u_k collected synchronously?
- Do the inputs arrive in the correct order through communication?
- What is the time duration between indices k and $k + 1$? Is it a constant?

These issues, addressing *where* and *when*, rather than *how*, to perform sensing, computation, and communication, play a central role in the overall system performance. However, these “nonfunctional” aspects of computation, related to concurrency, responsiveness, networking, and resource management, are not well supported by traditional programming models and languages. State-centric programming aims at providing design methodologies and frameworks that give meaningful abstractions for these issues, so that system designers can continue to write algorithms like (7.1) and (7.2) on top of an intuitive understanding of where and when the operations are performed. This section introduces one such abstraction, namely, collaboration groups.

7.5.1 Collaboration Groups

A collaboration group is a set of entities that contribute to a state update. These entities can be physical sensor nodes, or they can be more abstract system components such as virtual sensors or mobile agents hopping among sensors. In this context, they are all referred to as *agents*.

Intuitively, a collaboration group provides two abstractions: its *scope* to encapsulate network topologies and its *structure* to encapsulate communication protocols. The scope of a group defines the membership of the nodes with respect to the group. For the discussion of collaboration groups in this chapter, we broaden the notion of nodes to include both physical sensor nodes and virtual sensor nodes that may not be attached to any physical sensor. In this broader sense of node, a software agent that hops among the sensor nodes to track a target is a virtual node. Limiting the scope of a group to a subset of the entire space of all agents improves scalability. The scope of a group can be specified existentially or by a membership function (e.g., all nodes in a geometric extent, all nodes within a certain number of hops from an anchor node, or all nodes that are "close enough" to a temperature contour). Grouping nodes according to some physical attributes rather than node addresses is an important and distinguishing characteristic of sensor networks.

The *structure* of a group defines the "roles" each member plays in the group, and thus the flow of data. Are all members in the group equal peers? Is there a "leader" member in the group that consumes data? Do members in the group form a tree with parent and children relations? For example, a group may have a leader node that collects certain sensor readings from all followers. By mapping the leader and the followers onto concrete sensor nodes, we effectively define the flow of data from the hosts of followers to the host of the leader. The notion of roles also shields programmers from addressing individual nodes either by name or address. Furthermore, having multiple members with the same role provides some degree of redundancy and improves robustness of the application in the presence of node and link failures.

Formally, a group is a 4-tuple:

$$G = (A, L, p, R)$$

where

A is a set of agents;

L is a set of labels, called *roles*;

$p : A \to L$ is a function that assigns each agent a role;

$R \subseteq L \times L$ are the connectivity relations among roles.

Given the relations among roles, a group can induce a lower-level connectivity relation E among the agents, so that $\forall a, b \in A$, if $(p(a), p(b)) \in R$, then $(a, b) \in E$. For example, under this formulation, the leader-follower structure defines two roles, $L = \{\mathit{leader}, \mathit{follower}\}$, and a connectivity relation, $R = \{(\mathit{follower}, \mathit{leader})\}$, meaning that the follower sends data to the leader. Then, by specifying one leader agent and multiple follower agents within a geographical region (i.e., specifying a map p from a set of agents in A to labels in L), we have effectively specified that all followers send data to the leader without addressing the followers individually.

At run time, the scope and structural dynamics of groups are managed by group management protocols, which are highly dependent on the types of groups. A detailed specification of group management protocols is beyond the scope of this section. Some examples of these protocols are discussed here at a high level. Interested readers can refer to Chapter 3 for more detail.

Examples of Groups

Combinations of scopes and structures create patterns of groups that may be highly reusable from application to application. Here, we give several examples of groups, though by no means is it a complete list. The goal is to illustrate the wide variety of the kinds of groups, and the importance of mixing and matching them in applications.

Geographically Constrained Group. A geographically constrained group (GCG) consists of members within a prespecified geographical extent. Since physical signals, especially the ones from point targets, may propagate only to a limited extent in an environment, this kind of group naturally represents all the sensor nodes that can possibly "sense" a phenomenon. There are many ways to specify the geographic shape, such as circles, polygons, and their unions and intersections. A GCG can be easily established by geographically constrained flooding. Protocols such as Geocasting [117], GEAR [229], and Mobicast [102] may be used to support the communication among members even in the presence of communication "holes" in the region. A GCG may have a leader, which fuses information from all other members in the group.

***N*-hop Neighborhood Group.** When the communication topology is more important than the geographical extent, hop counts are useful to constrain group membership. An n-hop neighborhood group (n-HNG) has an anchor node and defines that all nodes within n communication hops are members of the group. Since it uses hop counts rather than Euclidean distances, local broadcasting can be used to determine the scope. Usually, the anchor node is the leader of the group, and the group may have a tree structure with the leader as the root to optimize for communication. If the leader's behavior can be decomposed into suboperations running on each node, then the tree structure also provides a platform for distributing the computation.

There are several useful special cases for n-HNG. For example, 0-HNG contains only the anchor node itself, 1-HNG comprises the one-hop neighbors of the anchor node, and ∞-HNG contains all the nodes reachable from the root. From this point of view, TinyDB [149] (as discussed in Chapter 6) is built on a ∞-HNG group.

Publish/Subscribe Group. A group may also be defined more dynamically, by all entities that can provide certain data or services, or that can satisfy certain predicates over their observations or internal states. A publish/subscribe group (PSG) comprises consumers expressing interest in specific types of data or services and producers that provide those data or services. Communication among members of a PSG may be established via rendezvous points, directory servers, or network protocols such as directed diffusion.

Acquaintance Group. An even more dynamic kind of group is the acquaintance group (AG), where a member belongs to the group because it was "invited" by another member in the group. The relationships among the members may not depend on any physical properties at the current time but may be purely logical and historical. A member may also quit the group without requiring permission from any other member. An AG may have a leader, serving as the rendezvous point. When the leader is also fixed on a node or in a region, GPSR [112], ad hoc routing trees, or directed diffusion types of protocols may facilitate the communication between the leader and the other members. An obvious use of this group is to monitor and control mobile agents from a base station. When all members in the group are mobile, there is no leader member, and any member may wish to communicate to one or more other members, the maintenance of connectivity among the group members can be nontrivial. The roaming hub (RoamHBA) protocol is an example of maintaining connectivity among mobile agents [67].

Using Multiple Types of Groups

Mixing and matching groups is a powerful technique for tackling system complexity by making algorithms much more scalable and resource efficient without sacrificing conceptual clarity. One may use highly tuned communication protocols for specific groups to reduce latency and energy costs.

There are various ways to compose groups. They can be composed in parallel to provide different types of input for a single computational entity. For example, in the target tracking problem in

Chapters 2 and 5, one may use a GCG to gather sensor measurements, while using a 1-HNG to select the potential next leader. Groups may also be composed hierarchically, such that a group (or its representative member) is contained by another group. For example, while using multiple groups to compute target trajectories, all tracking leaders of various targets may form a PSG with a base station to report the tracking result to.

7.5.2 PIECES: A State-Centric Design Framework

PIECES (Programming and Interaction Environment for Collaborative Embedded Systems) [141] is a software framework that implements the methodology of state-centric programming over collaboration groups to support the modeling, simulation, and design of sensor network applications. It is implemented in a mixed Java-Matlab environment.

Principals and Port Agents

PIECES comprises *principals* and *port agents*. Figure 7.16 shows the basic relations among principals and port agents.

A principal is the key component for maintaining a piece of *state*. Typically, a principal maintains state corresponding to certain aspects of the physical phenomenon of interest.[11] The role of a principal is to update its state from time to time, a computation corresponding to evaluating function f in (7.1). A principal also accepts other principals' queries of certain views on its own state, a computation corresponding to evaluating function g in (7.2).

To update its portion of the state, a principal may gather information from other principals. To achieve this, a principal creates port agents and attaches them onto itself and onto the other principals. A port agent may be an input, an output, or both. An output port

11 From a computational perspective, a port agent as an object certainly has its own state. But the distinction here is that the states of port agents are *not* about physical phenomena.

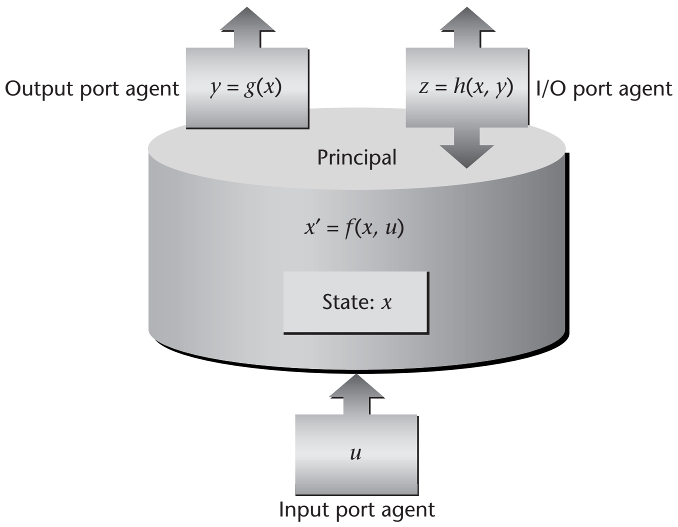


Figure 7.16 Principal and port agents (adapted from [141]).

agent is also called an *observer*, since it computes outputs based on the host principal's state and sends them to other agents. Observers may be active or passive. An active observer pushes data autonomously to its destination(s), while a passive observer sends data only when a consumer requests it. A principal typically attaches a set of observers to other principals and creates a local input port agent to receive the information collected by the remote agents. Thus port agents capture communication patterns among principals.

The execution of principals and port agents can be either time-driven or event-driven, where events may include physical events that are pushed to them (i.e., data-driven) or query events from other principals or agents (i.e., demand-driven). Principals maintain state, reflecting the physical phenomena. These states can be updated, rather than rediscovered, because the underlying physical states are typically continuous in time. How often the principal states need to be updated depends on the dynamics of the phenomena or physical events. The executions of observers, however, reflect the demands of the outputs. If an output is not currently needed, there is no need to compute it. The notion of "state" effectively separates these two execution flows.

To ensure consistency of state update over a distributed computational platform, PIECES requires that a piece of state, say $\mathbf{x}|_S$, can only be maintained by exactly one principal. Note that this does not prevent other principals from having local caches of $\mathbf{x}|_S$ for efficiency and performance reasons; nor does it prevent the other principals from locally updating the values of cached $\mathbf{x}|_S$. However, there is only one "master copy" for $\mathbf{x}|_S$; all local updates should be treated as "suggestions" to the master copy, and only the principal that owns $\mathbf{x}|_S$ has the final word on its values. This asymmetric access of variables simplifies the way shared variables are managed.

Principal Groups

Principals can form groups. A principal group gives its members a means to find other relevant principals and attaches port agents to them. A principal may belong to multiple groups. A port agent, however, serving as a proxy for a principal in the group, can only be associated with one group.

The creation of groups can be delegated to port agents, especially for leader-based groups. The leader port agent, typically of type input, can be created on a principal, and the port agent can take group scope and structure parameters to find the other principals and create follower port agents on them. Groups can be created dynamically, based on the collaboration needs of principals. For example, when a tracking principal finds that there is more than one target in its sensing region, it may create a classification group to fulfill the need of classifying the targets. A group may have a limited time span. When certain collaborations are no longer needed, their corresponding groups can be deleted.

The structure of a group allows its members to address other principals through their role, rather than their name or logical address. For example, the only interface that a follower port agent in a leader-follower structured group needs is to send data to the leader. If the leader moves to another node while a data packet is moving from a follower agent to the leader, the group management protocol should take care of the dangling packet, either delivering it to the leader at the new location or simply discarding it. The group management protocol may be built on top of data-centric routing and storage services such as diffusion routing and GHT (discussed in earlier chapters).

Mobility

A principal is hosted by a specific network node at any given time. The most primitive type of principal is a *sensing principal*, which is fixed to a sensor node. A sensing principal maintains a piece of (local) state related to the physical phenomenon, based solely on its own local measurement history. Although a sensing principal is constrained to a physical node, other principals may be implemented as software agents that move from host to host, depending on information utility, performance requirements, time constraints, and resource availability. A principal P may also be *attached* to another principal Q in the sense that P moves with Q. When a principal moves, it carries its state to the new location and the scope of the group it belongs to may be updated if necessary.

Mobile principals bring additional challenges to maintaining the state. For example, a principal should not move while it is in the middle of updating the state. To ensure this, PIECES imposes the restriction that whenever an agent is triggered, its execution must have reached a quiescent state. Such a trigger is called a *responsible trigger* [147]. Only at these quiescent states can principals move to other nodes in a well-defined way, carrying a minimum amount of information representing the phenomena.

PIECES Simulator

PIECES provides a mixed-signal simulator that simulates sensor network applications at a high level. The simulator is implemented using a combination of Java and Matlab. An event-driven engine is built in Java to simulate network message passing and agent execution at the collaboration-group level. A continuous-time engine is built in Matlab to simulate target trajectories, signals and noise, and sensor front ends. The main control flow is in Java, which maintains the global notion of time. The interface between Java and Matlab also makes it possible to implement functional algorithms such as signal processing and sensor fusion in Matlab, while leaving their execution control in Java. A three-tier distributed architecture is designed through Java registrar and RMI interfaces, so that the execution in Java and Matlab can be separately interrupted and debugged.

Like most network simulators such as ns-2, the PIECES simulator maintains a global event queue and triggers computational entities—principals, port agents, and groups—via timed events. However, unlike network simulators that aim to accurately simulate network behavior at the packet level, the PIECES simulator verifies CSIP algorithms in a networked execution environment at the collaboration-group level. Although groups must have distributed implementations in real deployments, they are centralized objects in the simulator. They can internally make use of instant access to any member of any role, although these services are not available to either principals or port agents. This relieves the burden of having to develop, optimize, and test the communication protocols concurrently with the CSIP algorithms. The communication delay is estimated based on the locations of sender and receiver and the group management protocol being used. For example, if an output port of a sensing principal calls `sendToLeader(message)` on its container group, then the group determines the sensor nodes that host the sensing principal and the destination principal, computes the number of hops between the two nodes specified by the group management protocol, and generates a corresponding delay and a bit error based on the number of hops. A detailed example of using this simulator is given in the next section.

7.5.3 Multitarget Tracking Problem Revisited

Using the state-centric model, programmers decouple a global state into a set of independently maintained pieces, each of which is assigned a principal. To update the state, principals may look for inputs from other principals, with sensing principals supporting the lowest-level sensing and estimation tasks. Communication patterns are specified by defining collaboration groups over principals and assigning corresponding roles for each principal through port agents. A mobile principal may define a utility function, to be evaluated at candidate sensor nodes, and then move to the best next location, all in a way transparent to the application developer. Developers can focus on implementing the state update functions as if they are writing centralized programs.

To make these concepts concrete, let us revisit the multitarget tracking system introduced in Chapter 2. Recall that in Figure 2.5, the tracking of two crossing targets can be decomposed into three phases:

1. When the targets are far apart, the tracking problem can be treated as a set of single-target tracking subproblems.
2. When the targets are in proximity of each other, they are tracked jointly due to signal mixing.
3. After the targets move apart, the tracking problem becomes two single-target tracking subproblems again.

To summarize, there are two kinds of target information that the user cares about in this context: target positions and target identities. In the third phase above, in addition to the problem of updating track locations, there is a need to sort out ambiguity regarding which track corresponds to which target. We refer to this problem as the *identity management* problem. Specifically, one must keep track of how the identities mix when targets cross over, and update identity information at the other node when credible target identity evidence is available to one node. The identity information may be obtained by a local classifier or by an identity management protocol across tracks. In PIECES, the system is designed as a set of communicating target trackers (`MTTrackers`), where each tracker maintains the trajectory and identity information about a target or a set of spatially adjacent targets. An `MTTracker` is implemented by three principals: a *tracking principal*, a *classification principal*, and an *identity management principal*, as shown in Figure 7.17. In the first phase, the identity state of the track is trivial; thus no classification and identity management principals are needed.

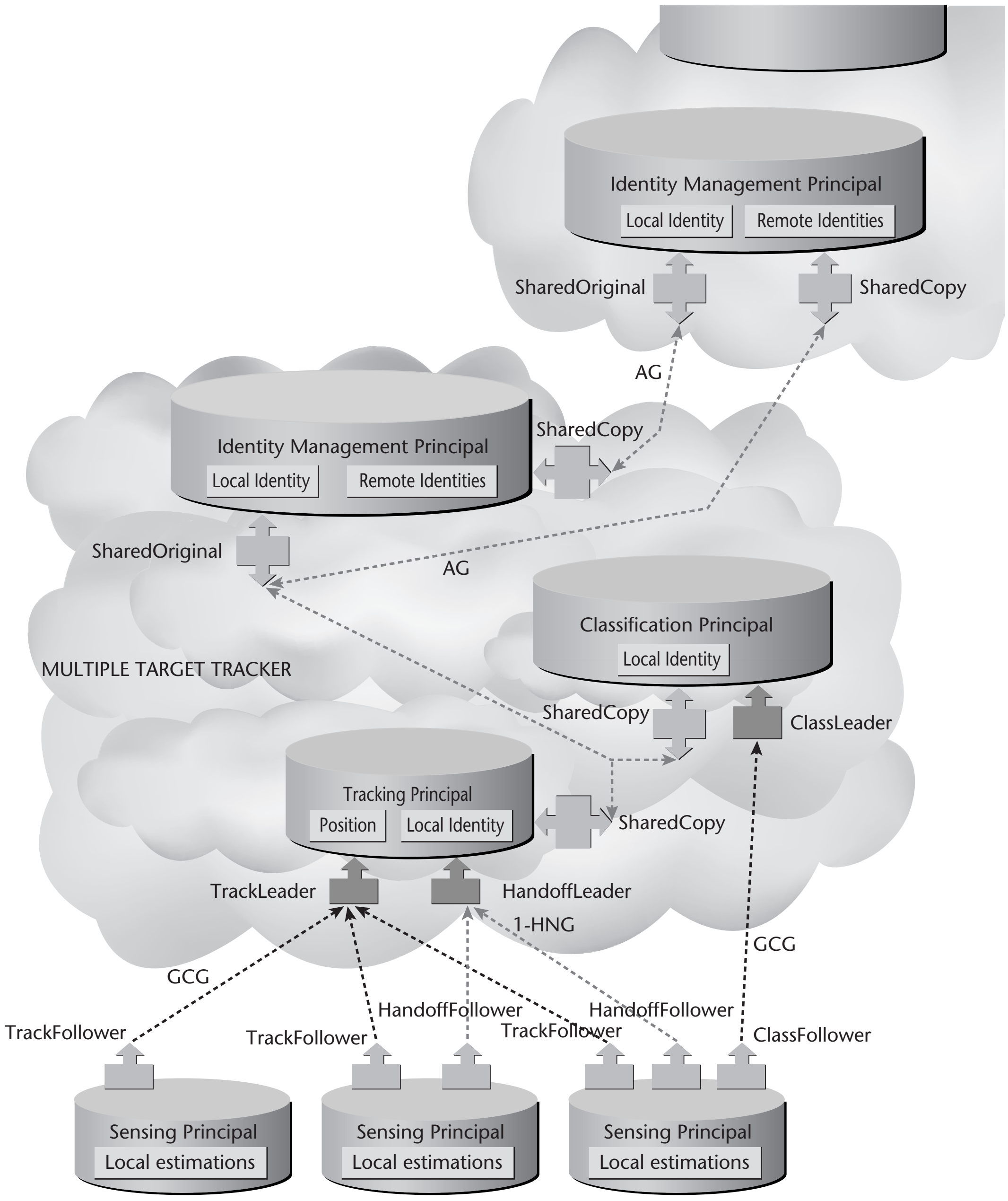


Figure 7.17 The distributed multi-object tracking algorithm as implemented in the state-centric programming model, using distributed principals and agents as discussed in the text. Notice that the state-centric model allows an application developer to focus on key pieces of state information the sensor network creates and maintains, thus raising the abstraction level of programming (adapted from [141]).

A tracking principal updates the track position state periodically. It collects local individual position estimates from sensors close to the target by a GCG with a leader-follower relation. The tracking principal is the leader, and all sensing principals within a certain geographical extent centered about the current target position estimate are the followers. The tracking principal also makes hopping decisions based on its current position estimate and the node characteristic information collected from its one-hop neighbors via a 1-HNG. When the principal is initialized, it creates the agents and corresponding groups. Behind the scene, the groups create follower agents with specific types of output, indicated by the sensor modalities. Without further instructions from the programmer, the followers periodically report their outputs to the input port agents. Whenever the leader principal is activated by a time trigger, it updates the target position using the newly received data from the followers and selects the next hosting node based on neighbor node characteristics.

Both the classification principal and the identity management principal operate on the identity state, with the identity management principal maintaining the "master copy" of the state. In fact, the classification principal is created only when there is a need for classifying targets. The classification principal uses a GCG to collect class feature information from nearby sensing principals in the same way that tracking principals collect location estimates. The identity management principal forms an AG with all other identity management principals that may have relevant identity information. They become members of a particular identity group only when targets intersect and their identities mix. Both classification principals and identity management principals are *attached* to the tracking principal for their mobility decisions. However, the formation of an AG among these three principals also provides the flexibility that they can make their own hopping decisions without changing their interaction interface.

Simulation Results

Figure 7.18 shows the progression of tracking two crossing targets. Initially, when the targets are well separated, as in Figure 7.18(a),

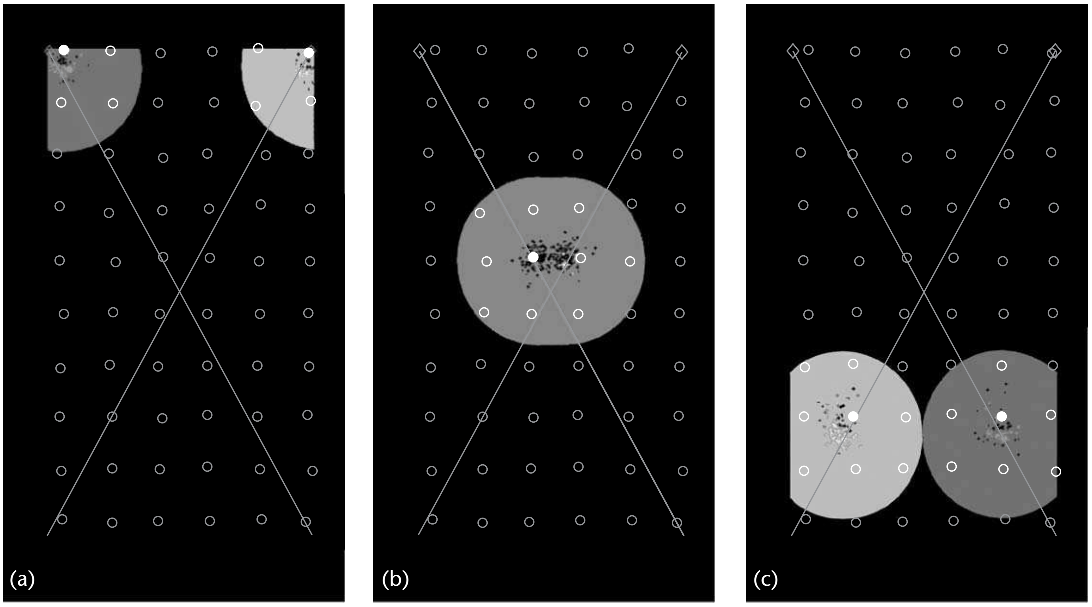


Figure 7.18 Simulation snapshots: Sensor nodes are indicated by small circles, and the crossing lines indicate the true trajectories of the two targets. One geographically constrained group is created for each target. When the two targets cross over, their groups merge into one.

each target is tracked by a tracker whose sensing group is pictured as a shaded disk. The hosting node of the tracking principal is plotted in solid white dots, and the hosts for corresponding sensing principals are plotted in small, empty white circles inside the shaded disks. Since the targets are well separated, each identity group contains only one member—the identity management principal of a tracker. As the targets move toward the center of the sensor field, the sensing groups move with their respective track positions. In Figure 7.18(b), the two separate tracking groups have merged. A joint tracking principal updates tracks for both targets. The reason for the merge is that when the two targets approach each other, it is more accurate to track the targets jointly, rather than independently, due to the effect of signal mixing. Finally, as the targets move away from each other, the merged tracking group splits into two separate single-target tracking groups that proceed to track each target separately, as shown in Figure 7.18(c). At this point, the identities of the targets are mixed, so that an identity group is created to contain the two identity management principals from both trackers.

Figure 7.19 shows a snapshot of a more complicated multitarget crossover scenario. Three tracks (A, B, and D) have crossed one another at some point in time; hence the identities of these tracks are mixed. The corresponding identity management principals form an identity group. Now, one identity management principal (the one at the bottom-right corner) collects classification information and identifies the track as belonging to the red target, as shown in the figure. Hence, it communicates with its peers in the identity management group (top-right and bottom-middle principals) to update the identity of their respective targets as well. The updated identity is shown in the right-hand-side bar chart in Figure 7.19. Defining the acquaintance group and its interface in this way allows these spatially distributed identity management principals to communicate with one another, thus providing the application developer the necessary abstraction for focusing on the functional aspect of identity management algorithms without worrying about the communication details.

7.6 Summary

This chapter has provided an overview of sensor network hardware and software platforms and application design methodologies. Although most of the existing platforms are tightly bound to particular hardware designs, the design principles covered in this chapter can be generalized to other hardware platforms as well. We described TinyOS, nesC, and TinyGALS, as examples of node-level operating systems and programming languages based on the Berkeley mote hardware. The node-centric platforms typically employ a message-passing abstraction of communication between nodes. Several networking protocols are covered in Chapter 3. Interfaces