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

TOOLS

CLASS : S6 ECE

SUBJECT CODE : CEC365

SUBJECT NAME : WIRELESS SENSOR NETWORKS DESIGN

REGULATION : 2021

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.

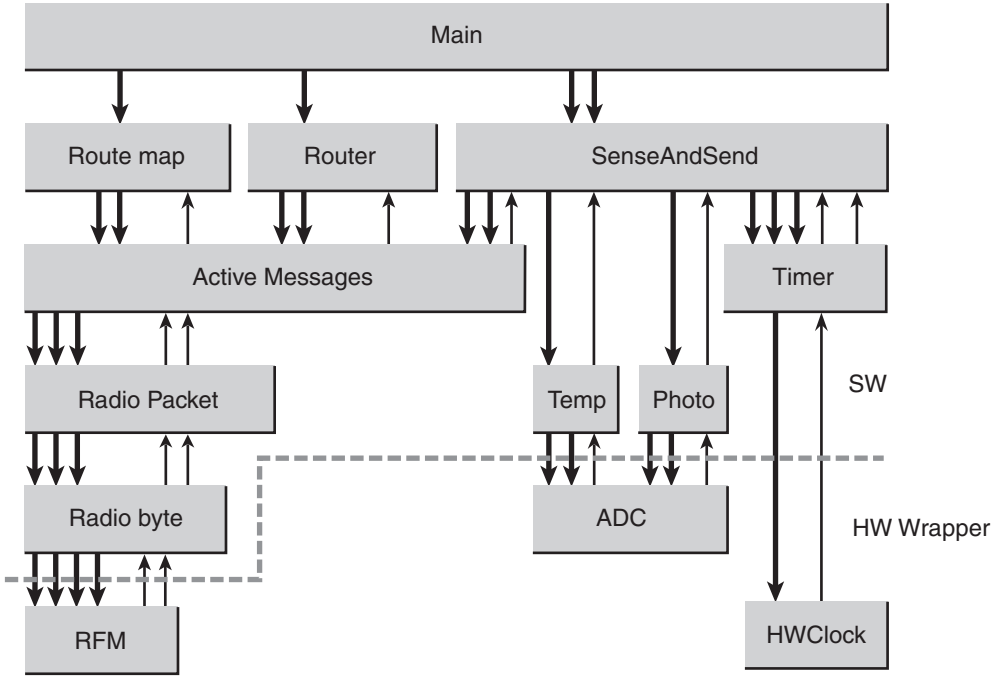

Figure 7.5 The `FieldMonitor` application for sensing and sending measurements.

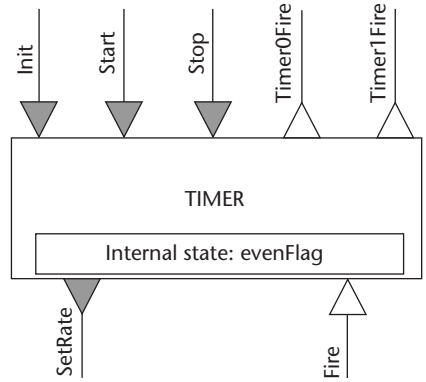

Figure 7.6 The `Timer` component and its interfaces.

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.

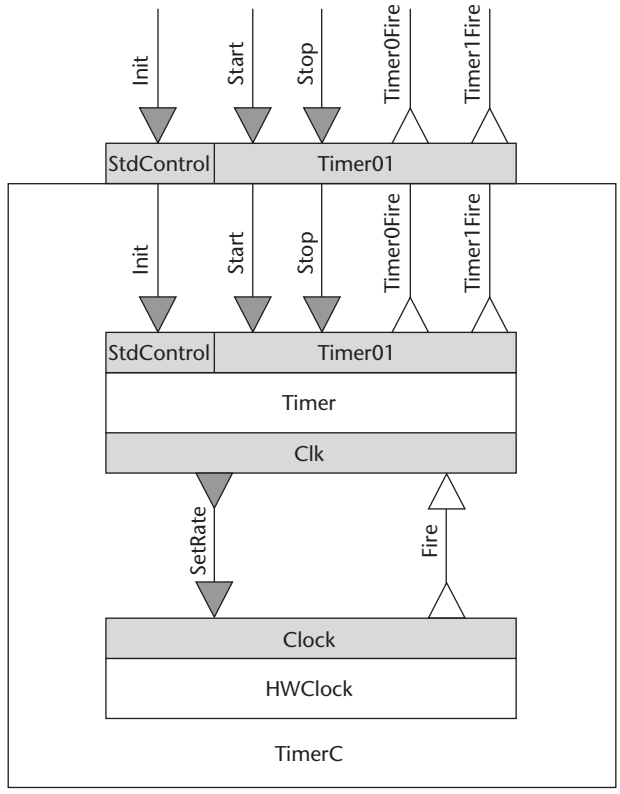

Figure 7.9 The `TimerC` configuration implemented by connecting `Timer` with `HWClock`.

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.timerOFire() {
    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 home-grown 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